Chombo Class LevelFluxRegisterEdge

Dan Martin (code mods) and Gustav Meglicki (weave)

November 29, 2006

Reflux.w,v 1.61 2006/11/29 16:46:39 gustav Exp

Contents

1 Copyright and Authors

Class **LevelFluxRegisterEdge** discussed herein is based on Chombo's **LevelFluxRegister**, code modifications by Dan Martin, LBNL, January 2000.

This weave is by Gustav Meglicki, Indiana University, for the Argonne National Laboratory. It was prepared with CWEB-3.64 by Silvio Levy and Donald E. Knuth, and LaTeX classes RCS-2.10 by Joachim Schrod and Jeffrey Goldberg and CWEB-3.6 by Joachim Schrod.

2 Introduction

This code implements a curl-refluxing correction on the boundary between fine and coarse levels.

Refluxing is a correction that is carried out at the very boundary, not around it, not inside it, but right on the faces of cells that are *shared* between the fine and coarse grids. So, the first place where this code differs markedly from the prolongation code discussed in the other document is in marking the boundary precisely, meaning that it is the actual faces we're going to mark. This will be done by selecting boundary boxes first and then also selecting the *sides* of the boxes that make the boundary. Two Chombo classes, **Side** and **SideIterator**, will be used in this game.

Fluid dynamic fluxes live on faces. In this they're quite like electromagnetic fields, which live on faces too. The meaning of Maxwell equations

$$c^2 \boldsymbol{\nabla} \times \boldsymbol{B} = \partial_t \boldsymbol{E} \tag{1}$$

$$\boldsymbol{\nabla} \times \boldsymbol{E} = -\partial_t \boldsymbol{B} \tag{2}$$

in their integral form—this also is how we treat Maxwell equations in computations—is that both $\boldsymbol{E}$ and $\boldsymbol{B}$ fields are associated with fluxes through the sides of the cell:

$$c^2 \oint_{\partial\,\text{side}} \boldsymbol{B} \cdot \mathrm{d}\boldsymbol{l} = \frac{\mathrm{d}}{\mathrm{d}t} \int_{\text{side}} \boldsymbol{E} \cdot \boldsymbol{n}\, \mathrm{d}s \tag{3}$$

$$\oint_{\partial\,\text{side}} \boldsymbol{E} \cdot \mathrm{d}\boldsymbol{l} = -\frac{\mathrm{d}}{\mathrm{d}t} \int_{\text{side}} \boldsymbol{B} \cdot \boldsymbol{n}\, \mathrm{d}s, \tag{4}$$

where $\boldsymbol{n}\,\mathrm{d}s$ is the surface differential pointing perpendicularly to the surface, and $\mathrm{d}\boldsymbol{l}$ is the edge differential pointing in the direction of the edge. $\partial\,$side means "the edge of the side". Using this notation we can also write that sides $= \partial\,$volume, which means that the sides are the edge of the volume—then side $\in \partial\,$volume.

But let us get back to fluid dynamics for a moment. When computations are carried out on two grids with different grid constants and different time steps, we may find that right on the boundary the coarse flux through the surface is different from the fine flux through the same surface. This difference in fluxes may result in a sharp field value step at the fine/coarse level boundary, that may later become source of noise and instability. The way to solve this problem is to replace the flux on the coarse side with the flux taken from the fine side or with a combination of the coarse and fine fluxes. This procedure is called *refluxing.* In computational electrodynamics refluxing is a little trickier. Here we have to add a special correction to the curl operator, so as to ensure that the field on the coarse side will remain divergence-free. This operation can be called *curl-refluxing.*

Standard Chombo implements fluid dynamics flux corrector as the **LevelFluxRegister** class. It is used as follows. The **LevelFluxRegister** :: *define* method creates two *flux registers* that are protected members of the class. One corresponds to the flux on the coarse side, the other one to the flux on the fine side. They are both fields of class **LevelData**⟨**FArrayBox**⟩ that cover the whole computational domain of their respective grids, even though only the boxes that correspond to the fine/coarse boundary are used. The method finds about the location of the boundary and boundary faces as well and stores this information on a protected variable *m_coarseLocations* that is a **LayoutData**⟨**Vector**⟨**Box**⟩⟩, where vector components correspond to the flux directions in a way that is reminiscent of **FluxBox**.

Two class methods **LevelFluxRegister** :: *incrementCoarse* and **LevelFluxRegister** :: *incrementFine* are used to fill *all* boxes of both registers with flux correction data. This is done box by box, which is why the methods take a box of flux corrections that the calling program has to calculate somehow, as their argument, not the whole **LevelData**. Normally we would scan over all boxes of a given level in a loop, calculate updates, fluxes and flux corrections for each of them and then, while still holding on to a box, we'd increment the corresponding register with the latter.

Once the registers have been filled, the refluxing is carried out by taking a combination of the original flux on the coarse side, the data accumulated in the coarse register and the data accumulated in the fine register and adding them up with some weights. This is done *only* in locations pointed to by *m_coarseLocations*.

The **LevelFluxRegisterEdge** works similarly, but is more elaborate in finding directions of faces on both the low and high sides of the fine/coarse grid boundary, because it deals with face and edge centered vector fields, not with cell centered scalars. It uses a more elaborate data structure called *refluxLocations*$[SpaceDim*2]$ of class **LayoutData**$\langle$**Vector**$\langle$**Vector**$\langle$**IntVectSet**$\rangle\rangle\rangle$, and a special map called *coarToCoarMap*$[SpaceDim*2]$ of class **LayoutData**$\langle$**Vector**$\langle$**DataIndex**$\rangle\rangle$ to characterize the boundary.

The curl-refluxing procedure uses averaged fine-side contour fields that have been accumulated in the registers. These are then used to correct the curl that yields the $\boldsymbol{B}$ (or $\boldsymbol{E}$) flux change on the coarse side.

To better explain what needs to be done here, we have to look more closely at what happens on the fine/coarse boundary in FDTD.

3 The Balsara Curl-Refluxing Procedure

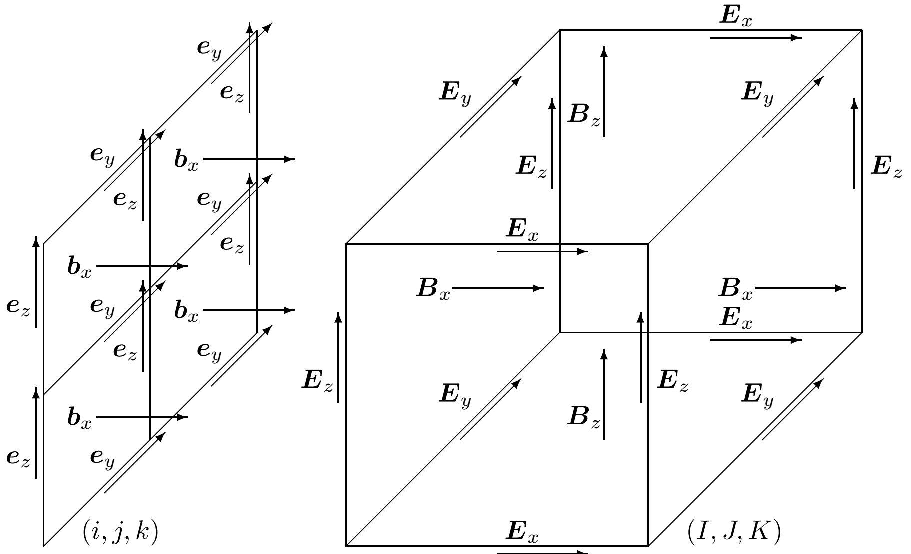

Figure 1: The interface between adjacent fine and coarse grids. The panel on the left shows a boundary fragment as seen from the fine grid side. The left side of the coarse grid cell number (I, J, K) shown on the right represents the same boundary fragment as seen from the coarse grid side.

Figure 1 shows the fine/coarse border wall as seen from both sides, from the fine side on the left and from the coarse side on the right. We assume here the refinement ratio of 2. Following Balsara [1], we will use capital letter symbols for all quantities on the right hand side, and small letter symbols for all quantities on the left hand side.

The coarse side view shows the whole coarse side cell that abuts the fine/coarse boundary. The left side of this cell is the boundary itself. If this cell is numbered by the triad (I, J, K), then the B_x arrow that punctures the left side of the cell is $B_x(I, J, K)$ as are the two $\boldsymbol{E}$ fields, E_z and E_y attached at the bottom of the side and at the edge that is closer to us. These are $E_y(I, J, K)$ and $E_z(I, J, K)$. The other E_y and E_z defined on this side correspond to the adjacent coarse grid cells, namely $E_y(I, J, K + 1)$ and $E_z(I, J + 1, K)$.

The other arrows attached to the other edges and sides of the cell are $B_x(I + 1, J, K)$, $B_z(I, J, K)$, $B_z(I, J, K + 1)$, $E_x(I, J, K)$, $E_x(I, J + 1, K)$, $E_x(I, J + 1, K + 1)$, $E_x(I, J, K + 1)$, $E_z(I + 1, J, K)$, $E_z(I + 1, J + 1, K)$, $E_y(I + 1, J, K)$, and $E_y(I + 1, J, K + 1)$. We have not drawn $B_y(I, J, K)$ and $B_y(I, J + 1, K)$ to avoid cluttering the figure, but they're there.

We should also remember that the $\boldsymbol{B}$ and $\boldsymbol{E}$ fields are shifted in time with respect to each other. If $\boldsymbol{B}$ is defined at time T and we're ready to advance it to $\boldsymbol{B}(T + \Delta T)$, then $\boldsymbol{E}$ is defined at time $T + \Delta T/2$.

To sum up, the symbols on the coarse grid side of the fine/coarse boundary are:

$$\begin{array}{l} B_x(I, J, K, T) \\ E_y(I, J, K, T + \Delta T/2) \\ E_z(I, J, K, T + \Delta T/2) \\ E_y(I, J, K + 1, T + \Delta T/2) \\ E_z(I, J + 1, K, T + \Delta T/2) \end{array}$$

The advance of $B_x(I, J, K, T) \rightarrow B_x(I, J, K, T + \Delta T)$ can now be read from equation (4). We have to look in the direction of B_x from the fine grid side and walk along the contour clockwise, from the bottom edge, for example, evaluating the left hand side of the equation at the same time. This results in

$$\begin{aligned} & E_y(I, J, K, T + \Delta T/2)\Delta Y + E_z(I, J + 1, K, T + \Delta T/2)\Delta Z \\ & \quad - E_y(I, J, K + 1, T + \Delta T/2)\Delta Y - E_z(I, J, K, T + \Delta T/2)\Delta Z \end{aligned} \tag{5}$$

On the right hand side we have the change of flux of B_x through the surface, which is

$$\frac{B_x(I,J,K,T+\Delta T) - B_x(I,J,K,T)}{\Delta T}\Delta Y \Delta Z \tag{6}$$

multiplied by -1. Equation (4) tells us that these should be equal:

$$\begin{aligned}
&E_y(I,J,K,T+\Delta T/2)\Delta Y + E_z(I,J+1,K,T+\Delta T/2)\Delta Z \\
&\quad -E_y(I,J,K+1,T+\Delta T/2)\Delta Y - E_z(I,J,K,T+\Delta T/2)\Delta Z \\
&\quad = -\frac{B_x(I,J,K,T+\Delta T) - B_x(I,J,K,T)}{\Delta T}\Delta Y \Delta Z
\end{aligned} \tag{7}$$

Dividing by $\Delta Y \Delta Z$ and rearranging terms on the left hand side yields

$$\begin{aligned}
&\frac{E_z(I,J+1,K,T+\Delta T/2) - E_z(I,J,K,T+\Delta T/2)}{\Delta Y} \\
&\quad -\frac{E_y(I,J,K+1,T+\Delta T/2) - E_y(I,J,K,T+\Delta T/2)}{\Delta Z} \\
&\quad = -\frac{B_x(I,J,K,T+\Delta T) - B_x(I,J,K,T)}{\Delta T},
\end{aligned} \tag{8}$$

which is a numerical approximation for

$$(\boldsymbol{\nabla} \times \boldsymbol{E})_x = \partial_y E_z - \partial_z E_y = -\partial_t B_x \tag{9}$$

This is how Stoke's theorem is explained to little children in country schools. The reason we do it here is to establish the notation, confirm that we get the signs and the directions right, and to prepare ground for stitching solutions on both sides of the fine/coarse border.

The time advance of B_x that follows from the above is

$$\begin{aligned}
&B_x(I,J,K,T+\Delta T) = B_x(I,J,K,T) \\
&\quad -\left(\frac{E_z(I,J+1,K,T+\Delta T/2) - E_z(I,J,K,T+\Delta T/2)}{\Delta Y}\right. \\
&\qquad \left. -\frac{E_y(I,J,K+1,T+\Delta T/2) - E_y(I,J,K,T+\Delta T/2)}{\Delta Z}\right)\Delta T.
\end{aligned} \tag{10}$$

This advance, when performed for $B_x(I,J,K,T)$, and similarly for $B_y(I,J,K,T)$ and $B_z(I,J,K,T)$ (see equations (18) and (19) below) at all (I,J,K) guarantees that the flux of $\boldsymbol{B}$ through all surfaces of cell (I,J,K) will remain zero at $T+\Delta T$, or, in other words, that there will be no divergence of $\boldsymbol{B}(I,J,K,T+\Delta T)$ *if* the same holds for $\boldsymbol{B}(I,J,K,T)$.

In numerical terms this evaluates to

$$\begin{aligned}
&\frac{B_x(I+1,J,K,T+\Delta T) - B_x(I,J,K,T+\Delta T)}{\Delta X} \\
&\quad +\frac{B_y(I,J+1,K,T+\Delta T) - B_y(I,J,K,T+\Delta T)}{\Delta Y} \\
&\quad +\frac{B_z(I,J,K+1,T+\Delta T) - B_z(I,J,K,T+\Delta T)}{\Delta Z} = 0,
\end{aligned} \tag{11}$$

for every (I,J,K) *if* the same holds for $\boldsymbol{B}(I,J,K,T)$ for every (I,J,K).

Now let us have a look at the fine grid side of the border. Here we see four sides of four cells and these are all *left* sides of the fine grid cells, the same it was with the coarse cell. These four fine grid cells live *inside* the coarse grid cell we have just looked at, in its left half. The fine grid cell that is on the bottom and closest to us is numbered by (i,j,k), where $i = 2I$, $j = 2J$ and $k = 2K$. The four b_x arrows correspond to $b_x(i,j,k)$, $b_x(i,j+1,k)$, $b_x(i,j+1,k+1)$ and $b_x(i,j,k+1)$.

The divergence-free data restriction procedure, which is carried out at every point in time whenever $\boldsymbol{b}$ and $\boldsymbol{B}$ are synchronized, replaces the flux of B_x through surface $(\Delta Y, \Delta Z)$ with the sum of the four fluxes of b_x through their matching surfaces, i.e.,

$$\begin{aligned}
&B_x(I,J,K,T)\Delta Y \Delta Z \\
&\quad \leftarrow b_x(i,j,k,T)\Delta y \Delta z + b_x(i,j+1,k,T)\Delta y \Delta z + b_x(i,j+1,k+1,T)\Delta y \Delta z + b_x(i,j,k+1,T)\Delta y \Delta z \\
&\quad = \left(b_x(i,j,k,T) + b_x(i,j+1,k,T) + b_x(i,j+1,k+1,T) + b_x(i,j,k+1,T)\right)\frac{\Delta Y}{2}\frac{\Delta Z}{2},
\end{aligned} \tag{12}$$

which, on dividing both sides by $\Delta Y \Delta Z$ reduces to replacing $B_x(I,J,K,T)$ with an arithmetic mean of the b_x fields defined on the fine grid sides the coarse cell side overlaps with:

$$B_x(I,J,K,T) \leftarrow \frac{1}{4}\left(b_x(i,j,k,T) + b_x(i,j+1,k,T) + b_x(i,j+1,k+1,T) + b_x(i,j,k+1,T)\right), \tag{13}$$

and similarly for $T + \Delta T$.

This restriction procedure guarantees that if the $\boldsymbol{b}$ field is divergence free everywhere on the fine grid, then $\boldsymbol{B}$ so produced will similarly be divergence free where it overlaps with $\boldsymbol{b}$, with the notable exception of the fine/coarse boundary, where we find that

$$\begin{aligned}
&\frac{B_x(I+1,J,K,T+\Delta T) - \frac{1}{4}\left(\begin{array}{c} b_x(i,j,k,T+\Delta T) + b_x(i,j+1,k,T+\Delta T) \\ +b_x(i,j+1,k+1,T+\Delta T) + b_x(i,j,k+1,T+\Delta T)\end{array}\right)}{\Delta X} \\
&\quad + \frac{B_y(I,J+1,K,T+\Delta T) - B_y(I,J,K,T+\Delta T)}{\Delta Y} \\
&\quad + \frac{B_z(I,J,K+1,T+\Delta T) - B_z(I,J,K,T+\Delta T)}{\Delta Z} \neq 0,
\end{aligned} \tag{14}$$

because the arithmetic mean of the b_x fields is not going to be equal to B_x that it replaces, and so the balance breaks.

Yet, the broken balance can be restored by adding corrections to $B_y(I,J,K,T+\Delta T)$, $B_y(I,J+1,K,T+\Delta T)$, $B_z(I,J,K,T+\Delta T)$ and $B_z(I,J,K+1,T+\Delta T)$. The corrections work by cancelling the change

$$B_x(I,J,K,T+\Delta T) - \frac{1}{4}\left(\begin{array}{c} b_x(i,j,k,T+\Delta T) + b_x(i,j+1,k,T+\Delta T) \\ +b_x(i,j+1,k+1,T+\Delta T) + b_x(i,j,k+1,T+\Delta T)\end{array}\right) \tag{15}$$

introduced into the divergence equation by the restriction procedure on the fine/coarse boundary face.

To figure out what the corrections should be, we should first evaluate the change, and then we'll have to decide how to assign its counterbalance to B_y and B_z respectively.

The physical meaning of the corrections is as follows. The restriction procedure that replaces $B_x(I,J,K,T+\Delta T)$ is equivalent to using more accurate data, namely, the fine grid electric fields, (e_x, e_y, e_z), to evaluate B_x. But the contours that advance B_y and B_z touch the fine/coarse boundary too, even if with one edge only. It turns out that if we replace the E_y and E_z fields on these shared edges with their fine grid versions e_y and e_z, in formulas used to advance B_y and B_z, we'll restore the balance to the divergence equation (11).

And this is the Balsara curl-refluxing procedure.

Before we go any further, it is clear that we will have to be extremely sensitive to directions and to the exact placement of the fine/coarse boundary and fields attached to it, when implementing the procedure in the code. This is why the class **LevelFluxRegisterEdge** is rather difficult to write, understand and to modify, if need be.

Now, let us go back to the formulas. We'll write down the updates for $B_y(I,J,K,T)$ and $B_z(I,J,K,T)$ quickly using equation (2):

$$\begin{aligned}
\partial_t B_y &= -\left(\boldsymbol{\nabla} \times \boldsymbol{E}\right)_y = -\left(\partial_z E_x - \partial_x E_z\right) && (16) \\
\partial_t B_z &= -\left(\boldsymbol{\nabla} \times \boldsymbol{E}\right)_z = -\left(\partial_x E_y - \partial_y E_x\right), && (17)
\end{aligned}$$

which discretize to

$$\begin{aligned}B_y(I,J,K,T+\Delta T) &= B_y(I,J,K,T)\\ &-\left(\frac{E_x(I,J,K+1,T+\Delta T/2)-E_x(I,J,K,T+\Delta T/2)}{\Delta Z}\right.\\ &\left.-\frac{E_z(I+1,J,K,T+\Delta T/2)-E_z(I,J,K,T+\Delta T/2)}{\Delta X}\right)\Delta T,\end{aligned} \tag{18}$$

where $E_z(I,J,K,T+\Delta T/2)$ lives on the fine/coarse boundary, and

$$\begin{aligned}B_z(I,J,K,T+\Delta T) &= B_z(I,J,K,T)\\ &-\left(\frac{E_y(I+1,J,K,T+\Delta T/2)-E_y(I,J,K,T+\Delta T/2)}{\Delta X}\right.\\ &\left.-\frac{E_x(I,J+1,K,T+\Delta T/2)-E_x(I,J,K,T+\Delta T/2}{\Delta Y}\right)\Delta T,\end{aligned} \tag{19}$$

where $E_y(I,J,K,T+\Delta T/2)$ lives on the fine/coarse boundary. Equations (10), (18) and (19) together advance the $\boldsymbol{B}$ field within the coarse grid.

The fields $E_z(I,J,K,T+\Delta T/2)$ in equation (18) and $E_y(I,J,K,T+\Delta T/2)$ in equation (19) will have to be additionally tweaked by the curl-refluxing correction.

Now, let us have a look at how the $\boldsymbol{b}$ fields are advanced between T and $T+\Delta T$. We can simply use equations (10), (18) and (19) here and just replace capital letters with low case letters. But there is one more difference: the time step is different. For the space refinement ratio of 2, the time refinement ratio has to be 3 to make FDTD work at all levels and to maintain the synchrony between the $\boldsymbol{E}$ and $\boldsymbol{e}$ fields and the $\boldsymbol{B}$ and $\boldsymbol{b}$ fields [2]. And so we have that $\Delta t = \Delta T/3$. Also, to advance the $\boldsymbol{b}$ fields from T to $T+\Delta T$ we will have to perform three time steps.

So, here they are for b_x:

$$\begin{aligned}b_x(i,j,k,T+\Delta T/3) &= b_x(i,j,k,T)\\ &-\left(\frac{e_z(i,j+1,k,T+\Delta T/6)-e_z(i,j,k,T+\Delta T/6)}{\Delta y}\right.\\ &\left.-\frac{e_y(i,j,k+1,T+\Delta T/6)-e_y(i,j,k,T+\Delta T/6)}{\Delta z}\right)\Delta T/3\end{aligned} \tag{20}$$

$$\begin{aligned}b_x(i,j,k,T+2\Delta T/3) &= b_x(i,j,k,T+\Delta T/3)\\ &-\left(\frac{e_z(i,j+1,k,T+\Delta T/2)-e_z(i,j,k,T+\Delta T/2)}{\Delta y}\right.\\ &\left.-\frac{e_y(i,j,k+1,T+\Delta T/2)-e_y(i,j,k,T+\Delta T/2)}{\Delta z}\right)\Delta T/3\end{aligned} \tag{21}$$

$$\begin{aligned}b_x(i,j,k,T+\Delta T) &= b_x(i,j,k,T+2\Delta T/3)\\ &-\left(\frac{e_z(i,j+1,k,T+5\Delta T/6)-e_z(i,j,k,T+5\Delta T/6)}{\Delta y}\right.\\ &\left.-\frac{e_y(i,j,k+1,T+5\Delta T/6)-e_y(i,j,k,T+5\Delta T/6)}{\Delta z}\right)\Delta T/3\end{aligned} \tag{22}$$

We can express $b_x(i,j,k,T+\Delta T)$ directly in terms of $b_x(i,j,k,T)$ and the three intervening curls as follows

$$b_x(i,j,k,T+\Delta T) = b_x(i,j,k,T)$$

$$
\begin{aligned}
&-\Bigg(\frac{e_z(i,j+1,k,T+\Delta T/6)-e_z(i,j,k,T+\Delta T/6)}{\Delta y}\\
&\qquad-\frac{e_y(i,j,k+1,T+\Delta T/6)-e_y(i,j,k,T+\Delta T/6)}{\Delta z}\Bigg)\Delta T/3\\
&-\Bigg(\frac{e_z(i,j+1,k,T+\Delta T/2)-e_z(i,j,k,T+\Delta T/2)}{\Delta y}\\
&\qquad-\frac{e_y(i,j,k+1,T+\Delta T/2)-e_y(i,j,k,T+\Delta T/2)}{\Delta z}\Bigg)\Delta T/3\\
&-\Bigg(\frac{e_z(i,j+1,k,T+5\Delta T/6)-e_z(i,j,k,T+5\Delta T/6)}{\Delta y}\\
&\qquad-\frac{e_y(i,j,k+1,T+5\Delta T/6)-e_y(i,j,k,T+5\Delta T/6)}{\Delta z}\Bigg)\Delta T/3
\end{aligned}
\tag{23}
$$

Now we are going to do the same for $b_x(i,j+1,k,T)$, $b_x(i,j+1,k+1,T)$, and $b_x(i,j,k+1,T)$:

$$
\begin{aligned}
&b_x(i,j+1,k,T+\Delta T)=b_x(i,j+1,k,T)\\
&-\Bigg(\frac{e_z(i,j+2,k,T+\Delta T/6)-e_z(i,j+1,k,T+\Delta T/6)}{\Delta y}\\
&\qquad-\frac{e_y(i,j+1,k+1,T+\Delta T/6)-e_y(i,j+1,k,T+\Delta T/6)}{\Delta z}\Bigg)\Delta T/3\\
&-\Bigg(\frac{e_z(i,j+2,k,T+\Delta T/2)-e_z(i,j+1,k,T+\Delta T/2)}{\Delta y}\\
&\qquad-\frac{e_y(i,j+1,k+1,T+\Delta T/2)-e_y(i,j+1,k,T+\Delta T/2)}{\Delta z}\Bigg)\Delta T/3\\
&-\Bigg(\frac{e_z(i,j+2,k,T+5\Delta T/6)-e_z(i,j+1,k,T+5\Delta T/6)}{\Delta y}\\
&\qquad-\frac{e_y(i,j+1,k+1,T+5\Delta T/6)-e_y(i,j+1,k,T+5\Delta T/6)}{\Delta z}\Bigg)\Delta T/3,
\end{aligned}
\tag{24}
$$

$$
\begin{aligned}
&b_x(i,j+1,k+1,T+\Delta T)=b_x(i,j+1,k+1,T)\\
&-\Bigg(\frac{e_z(i,j+2,k+1,T+\Delta T/6)-e_z(i,j+1,k+1,T+\Delta T/6)}{\Delta y}\\
&\qquad-\frac{e_y(i,j+1,k+2,T+\Delta T/6)-e_y(i,j+1,k+1,T+\Delta T/6)}{\Delta z}\Bigg)\Delta T/3\\
&-\Bigg(\frac{e_z(i,j+2,k+1,T+\Delta T/2)-e_z(i,j+1,k+1,T+\Delta T/2)}{\Delta y}\\
&\qquad-\frac{e_y(i,j+1,k+2,T+\Delta T/2)-e_y(i,j+1,k+1,T+\Delta T/2)}{\Delta z}\Bigg)\Delta T/3\\
&-\Bigg(\frac{e_z(i,j+2,k+1,T+5\Delta T/6)-e_z(i,j+1,k+1,T+5\Delta T/6)}{\Delta y}\\
&\qquad-\frac{e_y(i,j+1,k+2,T+5\Delta T/6)-e_y(i,j+1,k+1,T+5\Delta T/6)}{\Delta z}\Bigg)\Delta T/3,
\end{aligned}
\tag{25}
$$

and

$$\begin{aligned}
b_x(i,j,k+1,T+\Delta T) &= b_x(i,j,k+1,T) \\
&-\left(\frac{e_z(i,j+1,k+1,T+\Delta T/6)-e_z(i,j,k+1,T+\Delta T/6)}{\Delta y}\right. \\
&\qquad\left.-\frac{e_y(i,j,k+2,T+\Delta T/6)-e_y(i,j,k+1,T+\Delta T/6)}{\Delta z}\right)\Delta T/3 \\
&-\left(\frac{e_z(i,j+1,k+1,T+\Delta T/2)-e_z(i,j,k+1,T+\Delta T/2)}{\Delta y}\right. \\
&\qquad\left.-\frac{e_y(i,j,k+2,T+\Delta T/2)-e_y(i,j,k+1,T+\Delta T/2)}{\Delta z}\right)\Delta T/3 \\
&-\left(\frac{e_z(i,j+1,k+1,T+5\Delta T/6)-e_z(i,j,k+1,T+5\Delta T/6)}{\Delta y}\right. \\
&\qquad\left.-\frac{e_y(i,j,k+2,T+5\Delta T/6)-e_y(i,j,k+1,T+5\Delta T/6)}{\Delta z}\right)\Delta T/3. \qquad (26)
\end{aligned}$$

And now we add equations (23), (24), (25) and (26). This will result in numerous cancellations—as it should, because we are really heading towards a discretized form of the contour integral for the flux change $\Delta B_x \Delta Y \Delta Z$ but expressed in terms of the $\boldsymbol{e}$ fields rather than the $\boldsymbol{E}$ fields.

The cancellations of various $\boldsymbol{e}$ terms must all occur within the same time slice. We have 3 time slices here: $T+\Delta T/6$, $T+\Delta T/2$ and $T+5\Delta T/6$, and for everyone of them, the cancellations will occur in the same way. It is therefore enough to trace the cancellations within just one slice.

Let us take the first one. Looking at the e_z fields we see immediately that four of the e_z terms cancel out. They're the ones that correspond to the middle of the $\Delta Y \Delta Z$ plaquette, namely, $e_z(i,j+1,k)$ and $e_z(i,j+1,k+1)$. What is left is the e_z part of the contour integral around the $\Delta Y \Delta Z$ plaquette:

$$\begin{aligned}
-\big(&+e_z(i,j+2,k,T+\Delta T/6)+e_z(i,j+2,k+1,T+\Delta T/6) \\
&-e_z(i,j,k,T+\Delta T/6)-e_z(i,j,k+1,T+\Delta T/6)\big)\frac{\Delta T}{3\Delta y} \qquad (27)
\end{aligned}$$

Similarly for e_y, we get the cancellation of the $e_y(i,j,k+1)$ and $e_y(i,j+1,k+1)$ terms, which leaves

$$\begin{aligned}
-\big(&-e_y(i,j+1,k+2,T+\Delta T/6)-e_y(i,j,k+2,T+\Delta T/6) \\
&+e_y(i,j,k,T+\Delta T/6)+e_y(i,j+1,k,T+\Delta T/6)\big)\frac{\Delta T}{3\Delta z} \qquad (28)
\end{aligned}$$

In summary we get the following for the $T+\Delta T/6$ slice:

$$\begin{aligned}
-&\left(\frac{e_z(i,j+2,k,T+\Delta T/6)+e_z(i,j+2,k+1,T+\Delta T/6)-e_z(i,j,k,T+\Delta T/6)-e_z(i,j,k+1,T+\Delta T/6)}{\Delta y}\right. \\
&\left.-\frac{e_y(i,j+1,k+2,T+\Delta T/6)+e_y(i,j,k+2,T+\Delta T/6)-e_y(i,j,k,T+\Delta T/6)-e_y(i,j+1,k,T+\Delta T/6)}{\Delta z}\right)\frac{\Delta T}{3}
\end{aligned}$$

Now we repeat this for the other two time slices, add it all up, divide by 4 and end up with the replacement for $B_x(I,J,K,T+\Delta T)$ in terms of the $\boldsymbol{e}$ fields:

$$\begin{aligned}
&B_x(I,J,K,T+\Delta T) = B_x(I,J,K,T) \\
&-\frac{1}{2}\left(\frac{e_z(i,j+2,k,T+\Delta T/6)+e_z(i,j+2,k+1,T+\Delta T/6)-e_z(i,j,k,T+\Delta T/6)-e_z(i,j,k+1,T+\Delta T/6)}{\Delta Y}\right.
\end{aligned}$$

$$
\begin{aligned}
&-\frac{e_y(i,j+1,k+2,T+\Delta T/6)+e_y(i,j,k+2,T+\Delta T/6)-e_y(i,j,k,T+\Delta T/6)-e_y(i,j+1,k,T+\Delta T/6)}{\Delta Z}\\
&+\frac{e_z(i,j+2,k,T+\Delta T/2)+e_z(i,j+2,k+1,T+\Delta T/2)-e_z(i,j,k,T+\Delta T/2)-e_z(i,j,k+1,T+\Delta T/2)}{\Delta Y}\\
&-\frac{e_y(i,j+1,k+2,T+\Delta T/2)+e_y(i,j,k+2,T+\Delta T/2)-e_y(i,j,k,T+\Delta T/2)-e_y(i,j+1,k,T+\Delta T/2)}{\Delta Z}\\
&+\frac{e_z(i,j+2,k,T+5\Delta T/6)+e_z(i,j+2,k+1,T+5\Delta T/6)-e_z(i,j,k,T+5\Delta T/6)-e_z(i,j,k+1,T+5\Delta T/6)}{\Delta Y}\\
&-\frac{e_y(i,j+1,k+2,T+5\Delta T/6)+e_y(i,j,k+2,T+5\Delta T/6)-e_y(i,j,k,T+5\Delta T/6)-e_y(i,j+1,k,T+5\Delta T/6)}{\Delta Z}\\
&\Bigg)\frac{\Delta T}{3},
\end{aligned}
\tag{29}
$$

where we have absorbed $1/2$ into $\Delta Y = 2\Delta y$ and $\Delta Z = 2\Delta z$. The other $1/2$ that is in front of the large bracket can be absorbed also on recognizing that

$$
\frac{e_z(i,j+2,k,T+\Delta T/6)+e_z(i,j+2,k+1,T+\Delta T/6)}{2}
\tag{30}
$$

is simply the average of e_z on the far edge of the $\Delta Y \Delta Z$ plaquette, and similarly for the other terms.

Here we simplify our notation somewhat. The notation used so far has been explicit and precise, but extremely verbose, and this makes it hard to notice certain things. So we introduce the following symbol

$$
\langle e_z(i,j+2,\bar{k})\rangle|_{T+\Delta T/6} = \frac{e_z(i,j+2,k,T+\Delta T/6)+e_z(i,j+2,k+1,T+\Delta T/6)}{2},
\tag{31}
$$

where a bar above k indicates averaging over k and $k+1$, or, in case the refinement ratio n is other than 2, over $k, k+1, \ldots, k+n-1$. This new notation lets us rewrite the rather bulky equation (29) in a way that shows better what's what:

$$
\begin{aligned}
&B_x(I,J,K,T+\Delta T) = B_x(I,J,K,T)\\
&-\left(\frac{\langle e_z(i,j+2,\bar{k})\rangle - \langle e_z(i,j,\bar{k})\rangle}{\Delta Y} - \frac{\langle e_y(i,\bar{j},k+2)\rangle - \langle e_y(i,\bar{j},k)\rangle}{\Delta Z}\right)\Bigg|_{T+\Delta T/6}\frac{\Delta T}{3}\\
&-\left(\frac{\langle e_z(i,j+2,\bar{k})\rangle - \langle e_z(i,j,\bar{k})\rangle}{\Delta Y} - \frac{\langle e_y(i,\bar{j},k+2)\rangle - \langle e_y(i,\bar{j},k)\rangle}{\Delta Z}\right)\Bigg|_{T+\Delta T/2}\frac{\Delta T}{3}\\
&-\left(\frac{\langle e_z(i,j+2,\bar{k})\rangle - \langle e_z(i,j,\bar{k})\rangle}{\Delta Y} - \frac{\langle e_y(i,\bar{j},k+2)\rangle - \langle e_y(i,\bar{j},k)\rangle}{\Delta Z}\right)\Bigg|_{T+5\Delta T/6}\frac{\Delta T}{3}
\end{aligned}
\tag{32}
$$

This is quite easy to understand. Let us observe that j and $j+2$ correspond geographically to J and $J+1$ and k and $k+2$ correspond geographically to K and $K+1$. A comparison with equation (10) tells us that here we take an arithmetic average of three curls in place of just one curl, and each of these is expressed in terms of arithmetic averages of the fine grid $\boldsymbol{e}$ fields over the relevant edge.

The difference between equations (32) and (10) is what we should redistribute between ΔB_y and ΔB_z in order to restore the broken divergence balance in the coarse grid cell that abuts the boundary.

Let us now turn back to equations (18) and (19). As we have pointed out, $E_z(I,J,K,T+\Delta T/2)$ and $E_y(I,J,K,T+\Delta T/2)$ live on the fine/coarse boundary. So why not replace them with time and edge averages constructed from the $\boldsymbol{e}$ fields, the same way we did with B_x in equation (32)? The resulting correction would be

$$
\begin{aligned}
&B_y(I,J,K,T+\Delta T) \leftarrow\\
&\quad B_y(I,J,K,T+\Delta T) + \frac{E_z(I,J,K,T+\Delta T/2)}{\Delta X}\Delta T\\
&\quad -\frac{\langle e_z(i,j,\bar{k})\rangle|_{T+\Delta T/6}}{\Delta X}\frac{\Delta T}{3} - \frac{\langle e_z(i,j,\bar{k})\rangle|_{T+\Delta T/2}}{\Delta X}\frac{\Delta T}{3} - \frac{\langle e_z(i,j,\bar{k})\rangle|_{T+5\Delta T/6}}{\Delta X}\frac{\Delta T}{3},
\end{aligned}
\tag{33}
$$

where we have first subtracted $-E_z(I, J, K, T+\Delta T/2)\Delta T/\Delta X$, and then replaced it with the three $\boldsymbol{e}$ terms. This can be nicely gathered into

$$
\begin{aligned}
B_y(I,J,K,T+\Delta T) &\leftarrow B_y(I,J,K,T+\Delta T)\\
&-\left(\frac{\langle e_z(i,j,\bar{k})\rangle|_{T+\Delta T/6}+\langle e_z(i,j,\bar{k})\rangle|_{T+\Delta T/2}+\langle e_z(i,j,\bar{k})\rangle|_{T+5\Delta T/6}}{3}\right.\\
&\qquad\left.-E_z(I,J,K,T+\Delta T/2)\right)\frac{\Delta T}{\Delta X}.
\end{aligned}
\tag{34}
$$

And this is the celebrated Balsara curl-refluxing correction for B_y.

A similar correction for $B_z(I, J, K, T+\Delta T)$ is

$$
\begin{aligned}
B_z(I,J,K,T+\Delta T) &\leftarrow B_z(I,J,K,T+\Delta T)\\
&+\left(\frac{\langle e_y(i,\bar{j},k)\rangle|_{T+\Delta T/6}+\langle e_y(i,\bar{j},k)\rangle|_{T+\Delta T/2}+\langle e_y(i,\bar{j},k)\rangle|_{T+5\Delta T/6}}{3}\right.\\
&\qquad\left.-E_y(I,J,K,T+\Delta T/2)\right)\frac{\Delta T}{\Delta X}.
\end{aligned}
\tag{35}
$$

Now we have to check if equations (32), (34) and (35) together save the day, i.e., if they restore the equality that has been broken by equation (14).

We assume that equation (11) is satisfied at T and at $T+\Delta T$, the latter if no restriction and refluxing corrections have been carried out yet. All $\boldsymbol{B}$ fields in this expression are affected by the restriction and the refluxing procedures with the exception of $B_x(I+1, J, K, T+\Delta T)$. Equations (32), (34) and (35) address the changes to three of the fields, so we still have to write down refluxing changes made to $B_y(I, J+1, K, T+\Delta T)$ and $B_z(I, J, K+1, T+\Delta T)$. These are

$$
\begin{aligned}
B_y(I,J+1,K,T+\Delta T) &\leftarrow B_y(I,J+1,K,T+\Delta T)\\
&-\left(\frac{\langle e_z(i,j+2,\bar{k})\rangle|_{T+\Delta T/6}+\langle e_z(i,j+2,\bar{k})\rangle|_{T+\Delta T/2}+\langle e_z(i,j+2,\bar{k})\rangle|_{T+5\Delta T/6}}{3}\right.\\
&\qquad\left.-E_z(I,J+1,K,T+\Delta T/2)\right)\frac{\Delta T}{\Delta X}
\end{aligned}
\tag{36}
$$

and

$$
\begin{aligned}
B_z(I,J,K+1,T+\Delta T) &\leftarrow B_z(I,J,K+1,T+\Delta T)\\
&+\left(\frac{\langle e_y(i,\bar{j},k+2)\rangle|_{T+\Delta T/6}+\langle e_y(i,\bar{j},k+2)\rangle|_{T+\Delta T/2}+\langle e_y(i,\bar{j},k+2)\rangle|_{T+5\Delta T/6}}{3}\right.\\
&\qquad\left.-E_y(I,J,K+1,T+\Delta T/2)\right)\frac{\Delta T}{\Delta X}.
\end{aligned}
\tag{37}
$$

So, now we have to fold equations (32), (34), (35), (36) and (37) together into (11) and see what happens. This is not as tedious as it seems, because of the marvellous invention of the copy-and-paste technology… We obtain the following.

$$
\begin{aligned}
\frac{1}{\Delta X}\Bigg(&B_x(I+1,J,K,T+\Delta T)-B_x(I,J,K,T)\\
&+\left(\frac{\langle e_z(i,j+2,\bar{k})\rangle-\langle e_z(i,j,\bar{k})\rangle}{\Delta Y}-\frac{\langle e_y(i,\bar{j},k+2)\rangle-\langle e_y(i,\bar{j},k)\rangle}{\Delta Z}\right)\Bigg|_{T+\Delta T/6}\frac{\Delta T}{3}
\end{aligned}
$$

$$
\begin{aligned}
&+\left(\frac{\langle e_z(i,j+2,\bar{k})\rangle - \langle e_z(i,j,\bar{k})\rangle}{\Delta Y} - \frac{\langle e_y(i,\bar{j},k+2)\rangle - \langle e_y(i,\bar{j},k)\rangle}{\Delta Z}\right)\Bigg|_{T+\Delta T/2}\frac{\Delta T}{3} \\
&+\left(\frac{\langle e_z(i,j+2,\bar{k})\rangle - \langle e_z(i,j,\bar{k})\rangle}{\Delta Y} - \frac{\langle e_y(i,\bar{j},k+2)\rangle - \langle e_y(i,\bar{j},k)\rangle}{\Delta Z}\right)\Bigg|_{T+5\Delta T/6}\frac{\Delta T}{3}\Bigg) \\
&+\frac{1}{\Delta Y}\Bigg(B_y(I,J+1,K,T+\Delta T) \\
&\quad -\Bigg(\frac{\langle e_z(i,j+2,\bar{k})\rangle|_{T+\Delta T/6} + \langle e_z(i,j+2,\bar{k})\rangle|_{T+\Delta T/2} + \langle e_z(i,j+2,\bar{k})\rangle|_{T+5\Delta T/6}}{3} \\
&\qquad -E_z(I,J+1,K,T+\Delta T/2)\Bigg)\frac{\Delta T}{\Delta X} \\
&\quad -B_y(I,J,K,T+\Delta T) \\
&\quad +\Bigg(\frac{\langle e_z(i,j,\bar{k})\rangle|_{T+\Delta T/6} + \langle e_z(i,j,\bar{k})\rangle|_{T+\Delta T/2} + \langle e_z(i,j,\bar{k})\rangle|_{T+5\Delta T/6}}{3} \\
&\qquad -E_z(I,J,K,T+\Delta T/2)\Bigg)\frac{\Delta T}{\Delta X}\Bigg) \\
&+\frac{1}{\Delta Z}\Bigg(B_z(I,J,K+1,T+\Delta T) \\
&\quad +\Bigg(\frac{\langle e_y(i,\bar{j},k+2)\rangle|_{T+\Delta T/6} + \langle e_y(i,\bar{j},k+2)\rangle|_{T+\Delta T/2} + \langle e_y(i,\bar{j},k+2)\rangle|_{T+5\Delta T/6}}{3} \\
&\qquad -E_y(I,J,K+1,T+\Delta T/2)\Bigg)\frac{\Delta T}{\Delta X} \\
&\quad -B_z(I,J,K,T+\Delta T) \\
&\quad -\Bigg(\frac{\langle e_y(i,\bar{j},k)\rangle|_{T+\Delta T/6} + \langle e_y(i,\bar{j},k)\rangle|_{T+\Delta T/2} + \langle e_y(i,\bar{j},k)\rangle|_{T+5\Delta T/6}}{3} \\
&\qquad -E_y(I,J,K,T+\Delta T/2)\Bigg)\frac{\Delta T}{\Delta X}\Bigg). \qquad (38)
\end{aligned}
$$

The first thing we notice is that *all* terms involving the $\boldsymbol{e}$ fields cancel out! What is left is

$$
\begin{aligned}
&\frac{1}{\Delta X}\Bigg(B_x(I+1,J,K,T+\Delta T) - \Big(B_x(I,J,K,T) \\
&\qquad -\Delta T\Big(\frac{E_z(I,J+1,K,T+\frac{\Delta T}{2}) - E_z(I,J,K,T+\frac{\Delta T}{2})}{\Delta Y} - \frac{E_y(I,J,K+1,T+\frac{\Delta T}{2}) - E_y(I,J,K,T+\frac{\Delta T}{2})}{\Delta Z}\Big)\Big)\Bigg) \\
&+\frac{1}{\Delta Y}\left(B_y(I,J+1,K,T+\Delta T) - B_y(I,J,K,T+\Delta T)\right) \\
&+\frac{1}{\Delta Z}\left(B_z(I,J,K+1,T+\Delta T) - B_z(I,J,K,T+\Delta T)\right) \qquad (39)
\end{aligned}
$$

A comparison with equation (10) shows us that the top two lines of equation (39) are simply

$$
\frac{1}{\Delta X}\left(B_x(I+1,J,K,T+\Delta T) - B_x(I,J,K,T+\Delta T)\right) \qquad (40)
$$

and so we find that equation (39) is the same as the left hand side of equation (11).

Quod erat demonstrandum.

An important lesson that follows from the above is that the restriction procedure given by equation (13) *must* be followed by the curl-refluxing corrections on the fine/coarse boundary given by equations (34) and (35), otherwise non-zero divergence of $\boldsymbol{B}$ will be introduced on the coarse side of the boundary. Through the prolongation procedure then the non-zero divergence will be re-introduced back into the fine level, thus generating instability.

In a system such as FDTD, where both the $\boldsymbol{E}$ and $\boldsymbol{B}$ fields must remain divergence free, curl-refluxing of $\boldsymbol{E}$ must be carried out as well, every time coarse level $\boldsymbol{E}$ is corrected by the data restriction from the fine level $\boldsymbol{e}$. The sign of the correction will have to be meticulously checked, because it is not going to be the same as for the $\boldsymbol{B}$ field.

An implementation of corrections (34) and (35) must remember the state of the fine level electric fields on the boundary at three previous time steps, namely at $T + \Delta T/6$, $T + \Delta T/2$ and $T + 5\Delta T/6$. This is what the reflux registers are for. The *incrementFine* and *incrementCoarse* methods accumulate corrections generated at the intermediate fine time steps in the registers. The accumulated values are then used by *refluxCurl* to add the corrections to B_y and B_z when the fine and the coarse levels are synchronized.

Before we get down to talking about the implementation of the curl-refluxing procedure, let us address the issue of *signs* first.

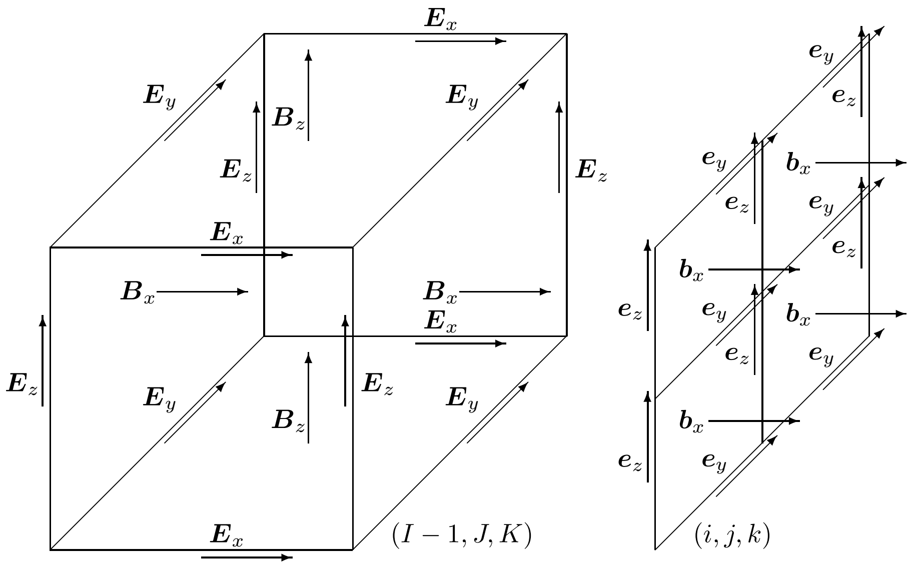

Figure 2: The interface between adjacent fine and coarse grids on the *low* side of the fine grid. The panel on the left shows the $(I-1,J,K)$ coarse grid cell that abuts the boundary with its *high* wall. The panel on the right shows the same fragment of the boundary, but seen from the fine grid side.

Figure 2 is much the same as Figure 1, but with the coarse and fine grids switched around. The coarse cell shown in the figure is $(I-1,J,K)$, not (I,J,K) as was the case in figure 1. The (I,J,K) cell in this case does not need any corrections. It is the $(I-1,J,K)$ cell where we need to reflux. Because this time the restricted B_x is on the high side of the cell, the signs change.

The B_y and B_z update equations on this side of the fine grid are

$$\begin{aligned} B_y(I-1,J,K,T+\Delta T) &= B_y(I-1,J,K,T) \\ &-\left(\frac{E_x(I-1,J,K+1,T+\Delta T/2) - E_x(I-1,J,K,T+\Delta T/2)}{\Delta Z}\right. \\ &\left.-\frac{E_z(I,J,K,T+\Delta T/2) - E_z(I-1,J,K,T+\Delta T/2)}{\Delta X}\right)\Delta T, \end{aligned} \tag{41}$$

where $E_z(I,J,K,T+\Delta T/2)$ lives on the boundary face, and

$$B_z(I-1,J,K,T+\Delta T) = B_z(I-1,J,K,T)$$

$$-\left(\frac{E_y(I,J,K,T+\Delta T/2)-E_y(I-1,J,K,T+\Delta T/2)}{\Delta X}\right.$$
$$\left.-\frac{E_x(I-1,J+1,K,T+\Delta T/2)-E_x(I-1,J,K,T+\Delta T/2}{\Delta Y}\right)\Delta T, \tag{42}$$

where $E_y(I,J,K,T+\Delta T/2)$ lives on the boundary face.

Therefore the curl-refluxing equations for $B_y(I-1,J,K,T+\Delta T)$ and for $B_z(I-1,J,K,T+\Delta T)$ on the low side are:

$$B_y(I-1,J,K,T+\Delta T) \leftarrow B_y(I-1,J,K,T+\Delta T)$$
$$+\left(\frac{\langle e_z(i,j+2,\bar{k})\rangle|_{T+\Delta T/6}+\langle e_z(i,j+2,\bar{k})\rangle|_{T+\Delta T/2}+\langle e_z(i,j+2,\bar{k})\rangle|_{T+5\Delta T/6}}{3}\right.$$
$$\left.-E_z(I,J,K,T+\Delta T/2)\right)\frac{\Delta T}{\Delta X} \tag{43}$$

and

$$B_z(I-1,J,K,T+\Delta T) \leftarrow B_z(I-1,J,K,T+\Delta T)$$
$$-\left(\frac{\langle e_y(i,\bar{j},k+2)\rangle|_{T+\Delta T/6}+\langle e_y(i,\bar{j},k+2)\rangle|_{T+\Delta T/2}+\langle e_y(i,\bar{j},k+2)\rangle|_{T+5\Delta T/6}}{3}\right.$$
$$\left.-E_y(I,J,K,T+\Delta T/2)\right)\frac{\Delta T}{\Delta X}. \tag{44}$$

Comparing equations (34) and (43) and then (35) and (44) we find that the sign of the correction on the low side is reversed.

4 Application of the Curl-Refluxing Correction

Instead of explaining the code in its code order, here we'll explain it back to front, focusing on the important routines first, while we're hot at it, and then only getting down to the tedium of various declarations.

The first method we're going to look at is **LevelFluxRegisterEdge**::*refluxCurl*, which implements equations (34) and (35). This will give us a better understanding of what various structures are for, before we get to build them with **LevelFluxRegisterEdge**::*define*.

When *refluxCurl* is called, we should already have terms such as

$$\left(\langle e_z(i,j,\bar{k})\rangle|_{T+\Delta T/6} + \langle e_z(i,j,\bar{k})\rangle|_{T+\Delta T/2} + \langle e_z(i,j,\bar{k})\rangle|_{T+5\Delta T/6}\right)/3 \tag{45}$$

accumulated in the B_y slots of the fine register *fabFine*, and terms such as

$$-E_z(I,J,K,T+\Delta T/2) \tag{46}$$

stored in the B_y slots of the coarse register *fabCoarse*. Similarly for B_z and B_x for all outward looking faces of the fine/coarse border. B_x too, because some faces will face in the z and in the y directions, in which case it will be the B_x field that will have to be curl-refluxed, together with the other field in the plane of the face.

The method takes two arguments. The first one is a coarse field *a_uCoarse*. This can be, for example, $\boldsymbol{B}$, with B_x, B_y, and B_z attached to appropriate sides of **FluxBox**es assembed into a **LevelData**. The second argument is *a_scale*. This is a number that will be used to multiply the sum of the registers.

We will encounter the following code expression within the innermost loop of the method:
$U(iv(\,), comp) \mathrel{-}= local\_scale * (coar(iv(\,), comp) + fine(iv(\,), comp))$. The *local_scale* parameter will turn out to be something like ΔT, and it will be made of *a_scale* with its sign adjusted depending on the side of the face and the component of the field to be corrected. $coar(iv(\,), comp)$ will be the coarse register content, $fine(iv(\,), comp)$ will be the fine register content and $U(iv(\,), comp)$ will be the field (B_x, B_y or B_z) being corrected. The structure of this expression fits exactly equations (34) and (35).

So, now that we know what the parameters are, let us have a look at the general outline of the method.

We begin with two simple sanity checks—first, if the whole **LevelFluxRegisterEdge** class has been fully defined, this is done by *assert*(*isDefined*( )), and, second, if the number of components in the input field *a_uCoarse* is the same as the number of components for which the object has been built. This number is stored on *m_nComp*.

Assuming it's safe to proceed, we go into four loops that scan over

1. the face directions: 0 (x), 1 (y), and 2 (z);
2. sides: *Lo* and *Hi*—for every face direction;
3. *all* boxes of the *a_uCoarse* **BoxLayout**—for every side of every face direction;
4. all field directions perpendicular to the direction of the face—for every box and for every side of every face direction.

At the innermost level of these four loops we have all four parameters defined: the box, the direction of the face, whether it is the low or the high face and the direction of the field to be corrected. This direction will be always perpendicular to the direction of the face.

There are two chunks here that we have left undefined. The first one, prior to entering the loop over all boxes, ⟨ Make a coarse copy of the fine register 5 ⟩, creates a copy of the fine register on the coarse register's grid. The second chunk, ⟨ Apply the correction 6 ⟩, checks if a given box has any locations that need to be refluxed, and if it does, it refluxes them.

```
4   ⟨ refluxCurl 4 ⟩ ≡
      void LevelFluxRegisterEdge::refluxCurl(LevelData⟨FluxBox⟩ &a_uCoarse, Real a_scale)
      {
        assert(isDefined( ));
        assert(a_uCoarse.nComp( ) ≡ m_nComp);
        SideIterator side;
```

```
  for (int idir = 0; idir < SpaceDim; ++idir) {
    for (side.begin(); side.ok(); ++side) {
      ⟨ Make a coarse copy of the fine register 5 ⟩
      for (DataIterator it = a_uCoarse.dataIterator(); it.ok(); ++it) {
        for (int fluxComp = 0; fluxComp < SpaceDim; fluxComp++) {
          if (fluxComp ≠ idir) {
            ⟨ Apply the correction 6 ⟩
          }
        }
      }
    }
  }
}
```
This code is used in chunk 39.

¶ The refluxing registers *fabFine* and *fabCoarse* are vectors of **LevelData**⟨**FluxBox**⟩es. Each has *SpaceDim* $*$ 2 slots, two slots that correspond to both sides per space dimension. Function *index*(*idir*, *side*()) retrieves the appropriate vector index given the face direction and the side, and this is then used to retrieve the corresponding **LevelData**⟨**FluxBox**⟩ that contains accumulated register entries for this direction and this side.

Both the fine and the coarse registers are defined on their own special box layouts that trace the boundaries of the coarsened fine level box layouts. Their boxes can be pointed to by the same **DataIterator**s and **LayoutIterator**s as the fine level boxes. But **DataIterator**s and **LayoutIterator**s that refer to the coarse level grid are different and a special translation must be used to switch from the coarse level iterator to the register iterator.

The construction of the fine register box layout, and there is a *different* layout for each slot of the register, is discussed in more detail in module ⟨ Iterate over Directions and Sides 30 ⟩.

The coarse register box layout is then used to define a new local register, called *fineRegLocal*, that is then filled with data from the fine register by the *copyTo* method.

The *crseCopiers* is a vector of **Copier**s that is indexed the same way as the registers. Its use here is only to speed things up by providing precomputed information to the *copyTo* method. A **Copier** is defined by specifying in advance the **BoxLayout**s between which a copy will take place and any other parameters that help specify the copy operation, for example, whether the ghost cells should be filled by the operation and if so, how many. In this case the **Copier**s of the *crseCopiers* vector are specified by **LevelFluxRegisterEdge**::*define* once the relevant **BoxLayout**s are known.

5 ⟨ Make a coarse copy of the fine register 5 ⟩ ≡
```
  LevelData⟨FluxBox⟩ &fineReg = fabFine[index(idir, side())];
  LevelData⟨FluxBox⟩ &coarReg = fabCoarse[index(idir, side())];
  LevelData⟨FluxBox⟩ fineRegLocal(coarReg.getBoxes(), m_nComp);

  fineReg.copyTo(fineReg.interval(), fineRegLocal, fineRegLocal.interval(), crseCopiers[index(idir, side())]);
```
This code is cited in chunk 4.
This code is used in chunk 4.

¶ Now let us have a look at the second chunk, ⟨ Apply the correction 6 ⟩. Table 1 lists some variables that are used within it, the first four of which would have been specified for this place within the loop.

We begin by extracting the field data for this box (*it*()) and for the field component (*fluxComp*) that is going to be refluxed in this iteration, from the input field *a_uCoarse*. The field will be referred to as U. For example, if *idir* points in the x direction, as is shown in Figure 1, then *fluxComp* will be y first and then z and U will be B_y first and then B_z—all restricted to this particular box.

We also extract the box on which the field is defined and call it *coarseGridBox*.

Now, the thing to know about this procedure, and this is not quite covered by Chombo documentation, is that if we were to extract a box from *a_uCoarse*, we'd extract a cell-centered box on which the **FluxBox** has

Table 1: Some variables used by the chunk.

name	meaning
idir	face direction (0, 1 or 2)
side( )	face side (*Lo* or *Hi*)
it( )	box pointer
fluxComp	direction of the field to be refluxed
a_uCoarse	the face-centered vector field, e.g., $\boldsymbol{B}$
a_scale	correction multiplication factor, e.g., ΔT
SpaceDim	number of space dimensions
index( )	index to the register vector
refluxLocations	a vector of cells for which refluxing should be done
coarToCoarMap	a map that converts a coarse level index to a coarse register index
coarReg	the coarse register
fineRegLocal	the fine register mapping onto the coarse register **BoxLayout**
nonUpdatedEdges	a list of edges that haven't been updated yet

been defined. But if instead we extract a specific field component from *a_uCoarse* first, as we do here, and this component is face-centered, and then only extract the box from the component, the box will come out face-centered and grown already in the direction of the field by one row (or column, or layer) of cells.

Consider the following simple 2-dimensional example. Suppose we have a cell centered **Box** $b(\mathbf{IntVect}::Zero, 5 * \mathbf{IntVect}::Unit)$. This box will show as

```
b: Box (0,0) to (5,5) type [(0,0)]
```

Now, let us build a **FluxBox** a on this box. There will be two fluxes associated with a (in two dimensions). Let us call them $B_x = a[0]$, and $B_y = a[1]$. We can now extract boxes from B_x and from B_y, as opposed to extracting a box from just a with *a.box*( ), which would merely return box b. The boxes that are extracted from the fluxes show as follows:

```
a[0].box(): Box (0,0) to (6,5) type [(1,0)]
a[1].box(): Box (0,0) to (5,6) type [(0,1)]
```

We can make the box $a[0].box(\,)$ cell centered but without changing its shape by invoking the **Box**::*shiftHalf*( ) method. For example:

```
Box c = a[0].box();
c.shiftHalf(0,1);
```

This will produce

```
c: Box (0,0) to (6,5) type [(0,0)]
```

The *shiftHalf*( ) operation, when performed in the +1 direction, affects only the **Box** type, stored in its own separate slot, but not its coordinate representation.

So, coming back to the code, we perform this *shiftHalf*( ) on *coarseGridBox* and then extract all cells from it and pack them into a set called *nonUpdatedEdges*.

The use of the *coarseGridBox* variable here seems redundant because we could have passed *U.box*( ) to the **IntVectSet** constructor directly with exactly the same effect. The **IntVectSet** constructor ignores the information about the data placement in cells. The variable *coarseGridBox* is not used anywhere else.

The discussion of this module continues in the next chunk.

6 ⟨Apply the correction 6⟩ ≡
FArrayBox &U = *a_uCoarse*[*it*( )][*fluxComp*];
Box *coarseGridBox* = *U.box*( );
coarseGridBox.shiftHalf(*fluxComp*, 1);

IntVectSet *nonUpdatedEdges*(*coarseGridBox*);

See also chunks 7, 8, and 9.
This code is cited in chunks 4, 6, and 9.
This code is used in chunk 4.

¶ The next step is to give the right sign to *a_scale*. We have shown above, equations (43) and (44), that the sign of the curl-refluxing correction gets reversed on the low side of the grid. Function *sign*(*side*( )) is defined on *LoHiSide.cpp* in the *BoxTools* subdirectory of the Chombo source and it returns −1 for the low side and +1 for the high side. So here we make use of it and define a correction coefficient *local_scale* as −*a_scale* for the high side and +*a_scale* for the low side, remembering that the actual correction is goin to be −*local_scale* ∗ (*fine_register* + *coarse_register*).

But the sign of the correction depends also on the direction of the face and the direction of the field being corrected with respect to the direction of the face. Looking again at equations (34) and (35) we find that

for B_x hi face: the B_y correction is with a minus and the B_z correction is with a plus;

for the other faces, we rotate everything cyclicly to find that

for B_y hi face: the B_z correction is with a minus and the B_x correction is with a plus;

for B_z hi face: the B_x correction is with a minus and the B_y correction is with a plus.

This logic is implemented by the ((*fluxComp* + 1) % (*SpaceDim*) ≡ *idir*) expression. It works as shown in Table 2. In other words we change the sign of *local_scale* again for B_z on the B_x face, for B_x on the B_y face and

Table 2: Value table for **if** ((*fluxComp* + 1)%(*SpaceDim*)) *local_scale* *= −1

face	idir	component	*fluxComp*	(*fluxComp* + 1) % 3	multiply by	side	*local_scale*
B_x	0	B_y	1	2	+1	lo	*a_scale*
						hi	−*a_scale*
		B_z	2	0	−1	lo	−*a_scale*
						hi	*a_scale*
B_y	1	B_z	2	0	+1	lo	*a_scale*
						hi	−*a_scale*
		B_x	0	1	−1	lo	−*a_scale*
						hi	*a_scale*
B_z	2	B_x	0	1	+1	lo	*a_scale*
						hi	−*a_scale*
		B_y	1	2	−1	lo	−*a_scale*
						hi	*a_scale*

for B_y on the B_z face. Remembering about the change of sign due to the side we find that, miraculously, we end up with just the right correction signs for all faces and for all directions of the refluxed fields.

The variable *testDir* is redundant. It is not used anywhere.

Another issue to ponder on is what if *SpaceDim* ≠ 3. The code is not developed to handle this case. There is an *assert*(*SpaceDim* ≡ 3) clause in **LevelFluxRegisterEdge**::*getRegComp*( ). That method is then called by **LevelFluxRegisterEdge**::*incrementCoarse*( ) and by **LevelFluxRegisterEdge**::*incrementFine*( ). Consequently a two-dimensional version of a Chombo program that tries to increment the **LevelFluxRegisterEdge** registers will abort if it has been compiled with the `DEBUG` flag.

The discussion of this module continues in the next chunk.

7 ⟨Apply the correction 6⟩ +≡
Real *local_scale* = −*sign*(*side*( )) ∗ *a_scale*;
int *testDir* = (*fluxComp* + 1) % (*SpaceDim*);
if (((*fluxComp* + 1) % (*SpaceDim*)) ≡ *idir*) *local_scale* ∗= −1;

¶ *refluxLocations* is an array of **LayoutData** defined by **LayoutData**⟨**Vector**⟨**Vector**⟨**IntVectSet**⟩⟩⟩ *refluxLocations*[*SpaceDim* ∗ 2]. The array entries correspond to directions and sides (hence the *SpaceDim* ∗ 2 size), i.e., there is an entry for the low side of the x direction and for the high side, and the same for y and z. Function *index*(*idir*, *side*( )) retrieves the corresponding slot. And so *refluxLocations*[*index*(*idir*, *side*( ))] is a **LayoutData** that corresponds to this face direction and to this side.

Now, what is this layout. Ultimately we will have a set of cells for which we need to do the refluxing assigned to each box. But if, say, the face is B_x, then we will have a separate set for B_y refluxing and a separate one for B_z refluxing, so we should have separate layouts for each *fluxComp*. And so, *refluxLocations*[*index*(*idir*, *side*( ))][*it*( )][*fluxComp*] is this set.

Because of the way the cells are collected, this set is further subdivided into slices that correspond to the fine grid boxes, which is why &*ivsV* is not a reference to **IntVectSet**, but a reference to **Vector**⟨**IntVectSet**⟩.

The same slices chop into portions a device that converts *it*( ), which points to a coarse grid box on which the field *a_uCoarse* is defined, to a corresponding vector of the coarse register boxes. The coarse register lives on top of the "coarsened" fine grid, so its numbering system and box layout is different from that of the coarse grid. A single coarse grid box may overlap with several fine grid boxes or coarse register boxes. This is where the slices originate from. The converter is called *coarToCoarMap*. Apart from its division into slices, there are different assignments for different directions *idir* and different sides.

The two instructions in this chunk extract the slice vectors for the *refluxLocations* and for the *coarToCoarMap* that correspond to the coarse grid box *it*( ), the direction and side of the face, and, in case of *refluxLocations* also the field component to be refluxed. The sets are then called *ivsV* and the maps *indexV*.

8 ⟨ Apply the correction 6 ⟩ +≡
Vector⟨**IntVectSet**⟩ &*ivsV* = *refluxLocations*[*index*(*idir*, *side*( ))][*it*( )][*fluxComp*];
Vector⟨**DataIndex**⟩ &*indexV* = *coarToCoarMap*[*index*(*idir*, *side*( ))][*it*( )];

¶ Finally we get to the curl-refluxing correction itself. We loop over all slices of the set of cells that need to be refluxed—for this coarse grid box, for this direction and side of the face and for this component *fluxComp*. For each slice we associate the set iterator *iv* with the set of cells in this slice. We find the corresponding register index, which is *indexV*[*i*] and make *coar* and *fine* point to the coarse and fine register portions that correspond to this field component in this box.

Now we begin iterating over the cells of the set. If the set is empty, then, of course, we don't do anything. This is going to be the case for the coarse grid cells that do not abut the fine/coarse boundary. If the set is not empty we apply the correction to U in cell *iv*( ) and to the component of U given by... *comp* (more about it below).

But let us observe first that the correction is applied with a minus, which reverses the signs of all entries in Table 2. We can, of course, define *a_scale* as being $-\Delta T$, or we can accumulate terms in the fine and the coarse registers with the opposite sign.

Whatever we do here, we must correlate it with what happens when we feed data into the registers. This is discussed in more detail in section 5.2.1, page 26.

The correction operation is enclosed in a loop over components *comp*. What is this business about? Isn't the component given by *fluxComp*? It is. *fluxComp* points to B_y or B_z if the face is B_x. But Chombo allows B_y and B_z to be multicomponent objects too. This can be used, for example, to group several B_y and B_z fields, or to combine B_y and E_y, or to have some higher rank tensor field defined on the **FluxBox**. But for a normal $\boldsymbol{B}$ field, we would have that *m_nComp* = 1. Because of the way **FluxBox** is defined, we have to address all *possible* components of each *flux*.

The **BaseFab** construct $U(iv(\,), comp)$ returns a modifiable lvalue reference to the *comp* component value defined at position *iv*( ) in the domain. The resulting assignment is a point operation, not a vector one.

In turn, the loop over the components *comp*, is enclosed in the **if** statement that executes the update only if the cell is still in the *nonUpdatedEdges* set. Let us recall that we have constructed this set at the beginning of the ⟨ Apply the correction 6 ⟩ module to contain all cells of the coarse grid box that corresponds to this *fluxComp*. As the updates are carried out, the cells are removed from the set by *nonUpdatedEdges* −= *iv*( ).

This way even if a given cell appears more than once in the *refluxLocations* sets, the corresponding update will not be repeated.

And this is it for *refluxCurl*.

9 ⟨Apply the correction 6⟩ +≡

```
IVSIterator iv;
for (int i = 0; i < ivsV.size(); ++i) {
  iv.define(ivsV[i]);
  const FArrayBox &coar = coarReg[indexV[i]][fluxComp];
  const FArrayBox &fine = fineRegLocal[indexV[i]][fluxComp];
  for (iv.begin(); iv.ok(); ++iv) {
    if (nonUpdatedEdges.contains(iv())) {
      for (int comp = 0; comp < m_nComp; ++comp)
        U(iv(), comp) -= local_scale * (coar(iv(), comp) + fine(iv(), comp));
      nonUpdatedEdges -= iv();
    }
  }
}
```

5 Managing the Registers

The **LevelFluxRegisterEdge** class provides a method for incrementing the fine register, *incrementFine*, and a method for incrementing the coarse register, *incrementCoarse*. These are used to accumulate terms such as $\langle e_z(i, j+2, \bar{k})\rangle/3$ at $t = T + \Delta T/6, T + \Delta T/2, T + 5\Delta T/6$ in the fine register and $-E_z(I, J, K, T)$ at $t = T + \Delta T/2$ in the coarse register.

The idea is that as we time step the system, as the required fields become available, we calculate the register contributions on the run and add them to whatever is in the register already—ultimately to use the accumulated content of the register when we are ready to apply the refluxing correction. For this to work, we must have one additional function, a function that wipes the registers clean before we restart the accumulation. The registers should be wiped immediately after the refluxing correction has been applied.

5.1 Wiping the Registers Clean

The function that cleans both registers is *setToZero*. And here we have its listing. The function is so simple that it requires little explanation. Let us recall that the registers are arrays of **LevelData**⟨**FluxBox**⟩, numbered by face directions and face sides. Given both, *index*(*idir*, *side*()) retrieves the appropriate slot. The function loops over all slots, then for each slot over all boxes of both registers and for each box it sets all flux-fields of the corresponding **FluxBox** to zero.

11 ⟨setToZero 11⟩ ≡

```
void LevelFluxRegisterEdge::setToZero()
{
  SideIterator side;
  for (int idir = 0; idir < SpaceDim; ++idir) {
    for (side.begin(); side.ok(); ++side) {
      LevelData<FluxBox> &fineReg = fabFine[index(idir, side())];
      LevelData<FluxBox> &coarReg = fabCoarse[index(idir, side())];
      for (DataIterator dit = fineReg.dataIterator(); dit.ok(); ++dit) fineReg[dit()].setVal(0.0);
      for (DataIterator dit = coarReg.dataIterator(); dit.ok(); ++dit) coarReg[dit()].setVal(0.0);
    }
  }
}
```

This code is used in chunk 39.

5.2 Incrementing the Coarse Register

The *incrementCoarse* method is the simpler one of the two, so we begin our discussion from it. Its arguments are as follows.

FArrayBox &*a_coarseFlux* This is an **FArrayBox**, *not* a **LevelData**. This is because *incrementCoarse* should be invoked when looping over the boxes of the coarse level to evaluate its fields. On this occasion, a user designed Fortran routine should be configured to evaluate contributions to the register as well. These should be then loaded into *a_coarseFlux* and passed to *incrementCoarse*.

Let us use equation (34) as an example.

In this case the coarse flux correction to be put in the register is simply $E_z(I, J, K, T + \Delta T/2)$ and the coarse flux to be corrected on the B_x face is B_y. The code here assumes that E_z is edge mounted, meaning that the **FArrayBox** type is (`NODE`, `NODE`, `CELL`).

The user should ensure that this is indeed the case.

Real *a_scale* This is the scaling factor, which in our example deriving from equation (34) could be set to $1/\Delta X$. If we do so then *a_scale* in the call to *refluxCurl* would be just ΔT.

DataIndex &*a_coarseDataIndex* This is the *it*( ) that we see in the Chombo **for** loops so often, that is, the object returned by the data iterator—the data index.

const Interval &*a_srcInterval* This is the source interval. Since **FArrayBox** may be a multicomponent object—there may be several E_z fields put together into it, or both $\boldsymbol{B}$ and $\boldsymbol{E}$ could be combined into a single **FArrayBox**—we need to specify from which slots to take the data.

const Interval &*a_dstInterval* Here we specify into which slots to write the data.

int *a_dir* This is the face direction.

The code assumes a convention that the flux entered into the coarse register at this stage should be a low side flux. We will discuss this and the sign convention in more detail in section 5.2.1, page 26.

12 ⟨incrementCoarse 12⟩ ≡
void LevelFluxRegisterEdge :: *incrementCoarse*(**FArrayBox** &*a_coarseFlux*, **Real** *a_scale*, **const DataIndex** &*a_coarseDataIndex*, **const Interval** &*a_srcInterval*, **const Interval** &*a_dstInterval*, **int** *a_dir*) {⟨Body of *incrementCoarse* 13⟩}

This code is used in chunk 39.

¶ The body of the method begins with sanity checks. We check if the object (of class **LevelFluxRegisterEdge**) is fully defined (and allocated) first. Then we check that the input flux, *a_coarseFlux*, is properly defined too, for example, that its box is not empty. We ensure that the direction of the flux to be corrected is well defined as well. It must be between 0 and $SpaceDim - 1$.

The number of slots on the destination side must be the same as the number of slots on the source side, and the number of slots must not exceed the number of components that the **LevelFluxRegisterEdge** object has been built for.

Note that all these sanity checks will work only if the code has been compiled with the `DEBUG` flag.

13 ⟨Body of *incrementCoarse* 13⟩ ≡

```
assert(isDefined());
assert(¬a_coarseFlux.box().isEmpty());
assert(a_dir ≥ 0);
assert(a_dir < SpaceDim);
assert(a_srcInterval.size() ≡ a_dstInterval.size());
assert(a_srcInterval.begin() ≥ 0);
assert(a_srcInterval.end() < a_coarseFlux.nComp());
assert(a_dstInterval.begin() ≥ 0);
assert(a_dstInterval.end() < m_nComp);
```

See also chunks 14 and 15.
This code is used in chunk 12.

¶ Next we find the direction of the $\boldsymbol{E}$ field passed to *incrementCoarse* in its first argument.

Consider again equation (34). Suppose the face is B_x and the component is B_y. The field residing on *a_coarseFlux* should be E_z (plus/minus the sign and the scaling factor).

As we have remarked above, the box extracted from the E_z field will be of type (NODE, NODE, CELL). The loop below skips the face direction x and then chooses the remaining direction to correspond to the CELL direction of the box, which is z. And so, *fluxComp* becomes z. This will work similarly for other cyclic permutations of x, y and z. In other words, the *fluxComp* component here ends up being the component of the electric field that's been passed to the function as its first argument.

Before leaving this part of the code we check that *fluxComp* has actually been assigned. We set it to -1 originally. The code will abort if it has been compiled with the DEBUG option and if *fluxComp* remains -1. This will happen if the box is of a wrong type.

14 ⟨ Body of *incrementCoarse* 13 ⟩ +≡

```
Box thisBox = a_coarseFlux.box();
int fluxComp = −1;
for (int sideDir = 0; sideDir < SpaceDim; sideDir++) {
      /* we do nothing in the direction normal to face */
  if (sideDir ≠ a_dir) {
    if (thisBox.type(sideDir) ≡ IndexType::CELL) {
      fluxComp = sideDir;
    }
  }
}
assert(fluxComp ≥ 0);
```

¶ And finally we get to the place where the register gets incremented. Function *getRegComp* is used to return the component of the register that will be incremented. This is the same component as the component of the $\boldsymbol{B}$ field that will be *corrected* by function *refluxCurl*. In the case of equation (34), *a_dir*, i.e., the face, is x, the component of the $\boldsymbol{E}$ field is z, which is also what *fluxComp* becomes, and the component of the field to be corrected should be y. And this is what *getRegComp* will return. It always returns the third component that is neither of the two given to it as arguments.

Now we iterate over the sides, i.e., we do things for *Lo* and *Hi* sides. For each *side*( ) and face direction *a_dir* we read the register box index, that corresponds to the **FArrayBox** box we're working on, from the *coarToCoarMap* and save it on *registerIndex*. This is not actually a single index, but the whole array of indexes, because a single coarse level box may overlap with several coarse register boxes.

We also retrieve the **LevelData**⟨**FluxBox**⟩ from the coarse register that corresponds to this face direction and side and call this *registers*.

Next we half-shift *a_coarseFlux* in the *a_dir* direction depending on which side we are on. For the low side we half-shift it by -1 and for the high side we half-shift it by $+1$.

As we have discussed above, a half-shift by $+1$ doesn't change the box coordinates, only its type. But a half-shift by -1 does change both the coordinates and the type of the box. For example, if we have a **FluxBox** field a defined on

```
b: Box (0,0) to (5,5) type [(0,0)]
```

then

```
a[0].box(): Box (0,0) to (6,5) type [(1,0)]
```

half-shifting this box in the 0 direction by $+1$ produces

```
a[0].box.shiftHalf(0,1): Box (0,0) to (6,5) type [(0,0)]
```

but half-shifting it in the 0 direction by -1 produces

```
a[0].box.shiftHalf(0,-1): Box (-1,0) to (5,5) type [(0,0)]
```

And this is exactly what we need on the low side, cf. equations (43) and (44).

Now we have to iterate over all coarse register boxes that may overlap with the given coarse level box, and the number of these is *registerIndex.size*( ). We extract the portion of the register that corresponds to this box (*registerIndex*[*i*]) and to this *regComp* direction, i.e., the direction of the $\boldsymbol{B}$ field, for which this correction is being collected, and simply add the *a_coarseFlux* field (i.e., E_z in our example), scaled by −*a_scale* (i.e., $-1/\Delta X$ in our example, watch the minus here) to it, for all slots between *a_srcInterval.begin*( ) and *a_srcInterval.begin*( ) + *a_srcInterval.size*( ). This is not a point operation. It is a vector operation, i.e., whole arrays are multiplied and then added together.

Having done so, we half-shift *a_coarseFlux* back into place in preparation for the next iteration over *side*s.

The **FArrayBox** :: *plus* operation works only on the intersection of the two **FArrayBox**es.

```
15 ⟨ Body of incrementCoarse 13 ⟩ +≡
  int regComp = getRegComp(a_dir, fluxComp);
  SideIterator side;
  for (side.begin( ); side.ok( ); ++side) {
    Vector⟨DataIndex⟩ &registerIndex = coarToCoarMap[index(a_dir, side( ))][a_coarseDataIndex];
    LevelData⟨FluxBox⟩ &registers = fabCoarse[index(a_dir, side( ))];
    a_coarseFlux.shiftHalf(a_dir, sign(side( )));
    for (int i = 0; i < registerIndex.size( ); ++i) {
      FluxBox &reg = registers[registerIndex[i]];
      FArrayBox &regDir = reg[regComp];
      regDir.plus(a_coarseFlux, −a_scale, a_srcInterval.begin( ), a_dstInterval.begin( ), a_srcInterval.size( ));
    }
    a_coarseFlux.shiftHalf(a_dir, −sign(side( )));
  }
```

5.2.1 The Saga of the Signs

There is a mind-boggling abundance of minuses in those places in the code, where the actual formulas are entered. So what is the ultimate sign on the actual expressions and what should we feed into *incrementCoarse* and then into *refluxCurl* to get the signs right?

Looking at equations (43) and (44) for the low side we see that we should eventually produce

$$-E_z(I+1, J, K, T+\Delta T/2)\frac{\Delta T}{\Delta X} \tag{47}$$

as the final coarse register correction to $B_y(I, J, K, T+\Delta T)$ and

$$+E_y(I+1, J, K, T+\Delta T)\frac{\Delta T}{\Delta X} \tag{48}$$

as the final coarse register correction to $B_z(I, J, K, T+\Delta T)$ on the low side B_x faces.

On the high side B_x faces we should have

$$+E_z(I, J, K, T+\Delta T/2)\frac{\Delta T}{\Delta X} \tag{49}$$

as the final coarse register correction to $B_y(I, J, K, T+\Delta T)$ and

$$-E_y(I, J, K, T+\Delta T/2)\frac{\Delta T}{\Delta X} \tag{50}$$

as the final coarse register correction to $B_z(I, J, K, T+\Delta T)$.

Suppose we call *incrementCoarse* with *a_coarseFlux* being $E_z(I, J, K, T+\Delta T/2)$ and mounted on a box of type (`NODE`, `NODE`, `CELL`), *a_scale* being $1/\Delta X$, and *a_dir* being x (or 0). Then, as we have already seen above, *fluxComp* ends up being z and *regComp* evaluates to y, which is the direction of the B_y field that should be corrected by *curlReflux*.

And so, the coarse register slot we end up operating on first is *fabCoarse*[*index*(x, *Lo*)][*box*][y], and then *fabCoarse*[*index*(x, *Hi*)][*box*][y]. Then there is this business with half-shifts, so that in the first case we end up adding

$$-\frac{1}{\Delta X}E_z(I+1,J,K,T+\Delta T/2) \tag{51}$$

to the register and in the second case

$$-\frac{1}{\Delta X}E_z(I,J,K,T+\Delta T/2). \tag{52}$$

Let us observe that the code does not change the sign of the contribution at this stage depending on the side. This will happen when we call *curlReflux*.

Table 3 sums up whats goes where for E_z either face or edge mounted.

Table 3: Coarse register assignment for a call on the B_x face with *a_coarseFlux* set to E_z and *a_scale* set to $1/\Delta X$.

register slot	register entry
fabCoarse[*index*(x, *Lo*)][*box*][y]	$-E_z(I+1,J,K,T+\Delta T/2)/\Delta X$
fabCoarse[*index*(x, *Hi*)][*box*][y]	$-E_z(I,J,K,T+\Delta T/2)/\Delta X$

So now, let us call *curlReflux* and see what happens there.

We call *curlReflux* with the coarse grid version of $\boldsymbol{B}$ and with its *a_scale* set to ΔT. What we are interested in is what is going to happen for *idir* $= 0$, i.e., x and *fluxComp* $= 1$, i.e., y. We are going to correct $B_y(I,J,K,T+\Delta T)$. From Table 2 we find that the *local_scale* factor for B_y on the B_x face is $+\Delta T$ for the low side, and $-\Delta T$ for the high side.

Let us focus on the low side. The actual slot we take from the coarse register is *fabCoarse*[*index*(x, *Lo*)][*box*][y], which contains $-E_z(I+1,J,K,T+\Delta T/2)/\Delta X$. *comp* in this case just stays 0, and the correction is $-$*local_scale* $*$ *fabCoarse*[*index*(x, *Lo*)][*box*][y], which evalutes to

$$+E_z(I+1,J,K,T+\Delta T/2)\frac{\Delta T}{\Delta X}, \tag{53}$$

so the sign comes out wrong.

This can be remedied by calling *incrementCoarse* with $-E_z$ and $+1/\Delta X$ or with $+E_z$ and $-1/\Delta X$. The general rule is that the sign of the field fed into the coarse register should be such as is required on the low side of the face.

5.3 Incrementing the Fine Register

Let us have a look again at equation (43) on page 15. The role of *incrementFine* is to accumulate terms such as

$$\frac{\langle e_z(i,j+2,\bar{k})\rangle|_{T+\Delta T/6}}{3}\frac{1}{\Delta X} = \frac{\left(e_z(i,j+2,k,T+\Delta T/6)+e_z(i,j+2,k+1,T+\Delta T/6)\right)/2}{3}\frac{1}{\Delta X} \tag{54}$$

for every fine level time step that fits within a given coarse level time step. We will see that this original version of *incrementFine* makes an assumption in one place that will break our FDTD fine level time step ($\Delta t = \Delta T/3$), but this is easy to change and both the location and the required change will be flagged.

The function takes the following arguments

FArrayBox &*a_fineFlux* This is the fine flux. For instance, given the above example, the e_z field

Real *a_scale* This is the scaling factor. For instance, given the above example, this could be $1/(3\Delta X)$. But analyzing the signs further down we will discover that we should set it to $-1/(3\Delta X)$.

DataIndex &*a_fineDataIndex* This is the pointer to the box of the fine level, that *a_fineFlux* lives on, as obtained by *it*( ) when iterating over the boxes of the fine level *DataLayout*. This function should be used similarly to how *incrementCoarse* is used. For every box of the fine level we operate on, we should evaluate *a_fineFlux* as well, using a specially designed Fortran routine and then pass the result to *incrementFine*.

Interval &*a_srcInterval* As was the case with *incrementCoarse*, this function can be used for a multiple-slot version of e_z, if we have more than one. So this variable specifies the interval from which we take the data.

Interval &*a_dstInterval* And this variable specifies the interval into which we intend to write the data.

int *a_dir* This is the face direction.

**Side :: ***LoHiSide a_sd* Here we specify the side (low or high) for which we do all this—meaning that when this function is called it will have to be called twice, for the low side first and then for the high side, explicitly.

17 ⟨ incrementFine 17 ⟩ ≡

void LevelFluxRegisterEdge :: *incrementFine* (**FArrayBox** &*a_fineFlux*, **Real** *a_scale*, **const DataIndex** &*a_fineDataIndex*, **const Interval** &*a_srcInterval*, **const Interval** &*a_dstInterval*, **int** *a_dir*, **Side** :: *LoHiSide a_sd*) {⟨ Body of incrementFine 18 ⟩}

This code is used in chunk 39.

¶ The body of the method begins with some sanity checks. First we make sure that the register is fully defined. Then we check that *a_fineFlux* is properly defined too, in particular, that it does not sit on an empty box. We ensure that we have the same number of slots both on the input and on the output objects, and that the slot intervals fit within the maximum slot intervals for both objects.

Then we make sure that the face direction is correct as well, i.e., within the limit of the space dimensionality and that the side is restricted to either low or high.

18 ⟨ Body of incrementFine 18 ⟩ ≡

```
assert(isDefined());
assert(¬a_fineFlux.box().isEmpty());
assert(a_srcInterval.size() ≡ a_dstInterval.size());
assert(a_srcInterval.begin() ≥ 0);
assert(a_srcInterval.end() < a_fineFlux.nComp());
assert(a_dstInterval.begin() ≥ 0);
assert(a_dstInterval.end() < m_nComp);
assert(a_dir ≥ 0);
assert(a_dir < SpaceDim);
assert((a_sd ≡ Side::Lo) ∨ (a_sd ≡ Side::Hi));
```

See also chunks 19, 20, 21, 22, 23, and 24.

This code is used in chunk 17.

¶ Here *a_scale* is additionally divided by the refinement ratio. This corresponds, as we will see later, to taking the average that we have marked with the bar in equation (54), i.e., $\bar{k}$. When fully expanded on the right hand side we find that the bracketed expression in the nominator is divided by 2. This 2 is *m_nRefine*.

For `CH_SPACEDIM` ≡ 2 this whole code will not work, so it would be best to flag this option as leading to error as well.

19
```
⟨Body of incrementFine 18⟩ +≡
#if (CH_SPACEDIM ≡ 2)
  Real denom = 1;
#elif (CH_SPACEDIM ≡ 3)
  Real denom = m_nRefine;
#else
  bogus spacedim;
#endif
  Real scale = a_scale/denom;
```

¶ This part of the code is *identical* to what we have seen in *incrementCoarse*, section 5.2. Here we find, again, the direction of the ***e*** field passed to *incrementFine* in its first argument. As was the case in *incrementCoarse* we expect that ***e*** will be edge mounted, which means that, for example, the e_z component will live in a box of type (`NODE`, `NODE`, `CELL`). So here we extract the box from the field, loop over all possible directions, with the exception of the face direction *a_dir* (because the ***e*** fields should live on its edges), and then choose the direction that corresponds to `CELL`. This becomes the *fluxComp* direction. If no such direction has been found, it means that the first argument is incorrectly sized and the program aborts,

but *only* if it's been compiled with the `DEBUG` option.

20
```
⟨Body of incrementFine 18⟩ +≡
  Box thisBox = a_fineFlux.box( );
  int fluxComp = −1;
  for (int sideDir = 0; sideDir < SpaceDim; sideDir ++) {
    if (sideDir ≠ a_dir) {
      if (thisBox.type(sideDir) ≡ IndexType::CELL) {
        fluxComp = sideDir;
      }
    }
  }
  assert(fluxComp ≥ 0);
```

¶ The logic of *incrementFine* is somewhat different from the logic of *incrementCoarse*, but similarities do extend at least to this chunk, where we find, as we did in *incrementCoarse*, the component of the ***B*** field to be corrected—this is the component that is different from both *a_dir* and *fluxComp*.

Then we retrieve the portion of the fine register that corresponds to (1) this fine level box, (2) this face direction, (3) this side, and (4) this component of the ***B*** field to be corrected, and call this *reg*. This is an **FArrayBox**, which may already contain some valid data, and to which we are going to add more.

Let us note here that the fine register is defined on the same grid as the fine level. This is why we do not need to use a translation map to switch from *a_fineDataIndex* to the register index, as we had to do in *incrementCoarse*. Here it is the same index.

Last, but not least, we half-shift the input *a_fineFlux* in the face direction either to the left or to the right depending on the side. After we have fed the data into the register, *a_fineFlux* will be shifted back.

21
```
⟨Body of incrementFine 18⟩ +≡
  int regcomp = getRegComp(a_dir, fluxComp);
  FluxBox &thisReg = fabFine[index(a_dir, a_sd)][a_fineDataIndex];
  FArrayBox &reg = thisReg[regcomp];
  a_fineFlux.shiftHalf(a_dir, sign(a_sd));
```

¶ Here we have a new part of the code that we haven't seen in *incrementCoarse*, but we have seen similar devices in **PiecewiseLinearFillPatchFace**. We are extracting here a box from *m_domainCoarse*, which we will then use to screen out cells that protrude outside the physical domain.

The operator &= modifies the box on the left by intersecting it with the argument box. **Box**::*grow* grows it by two cells, in this case, in each direction. **Box**::*surroundingNodes*(*regcomp*) converts the box to (`CELL`, `NODE`, `CELL`) if *regcomp* is y. This makes the box larger too, because it incorporates the *Hi* boundary of it in the *regcomp* direction.

Growing the box first and then intersecting it with its previous ungrown version accomplishes nothing. The box returns to the ungrown version.

Finally we intersect the register box with *shiftedValidDomain* and restrict the **BoxIterator** to it. This is where we chop off *reg.box*( ) cells that may protrude beyond the outer boundary of the coarse domain.

22 ⟨ Body of incrementFine 18 ⟩ +≡
Box *shiftedValidDomain* = *m_domainCoarse.domainBox*( );
shiftedValidDomain.grow(2);
shiftedValidDomain &= *m_domainCoarse*;
shiftedValidDomain.surroundingNodes(*regcomp*);
BoxIterator *regIt*(*reg.box*( ) & *shiftedValidDomain*);

¶ This double **for** loop is the heart of *incrementFine*.

We loop over the cells of the fine register box, since this is what *regIt*( ) points to. But this is a box of the *coarsened* fine grid, which means that we skip every second cell of the box if *m_nRefine* is 2. So now we are going to recover those sub-cells.

We create a single cell box that corresponds to the *coarseIndex* = *regIt*( ). This is going to be a (`CELL`, `CELL`, `CELL`) box. We then switch it to `NODE` in the *regcomp* direction. Invoking the example used above, equation (54), we would have *a_dir* = 0 (i.e., x), *fluxComp* = 2 (i.e., z), and *regcomp* = 1 (i.e., y), so the box would end up being (`CELL`, `NODE`, `CELL`). Now we create a refinement vector $(2, 2, 2)$ and refine the box.

This will refine the box in the directions perpendicular to *regcomp*. For example, if *coarseIndex* is $(4, 5, 6)$, then single cell box would be

```
Box (4,5,6) to (4,5,6) type [(0,0,0)]
```

After the *shiftHalf* operation the box would become

```
Box (4,5,6) to (4,5,6) type [(0,1,0)]
```

and after the *refine*

```
Box (8,10,12) to (9,10,13) type [(0,1,0)]
```

The reason for this is that the *refine* operation works differently on `CELL` centered cells and differently on `NODE` centered ones.

The box is then grown depending on whether we are on the low or on the high side. Our example box discussed above will become

```
Box (9,10,12) to (9,10,13) type [(0,1,0)]
```

on the low side and

```
Box (8,10,12) to (8,10,13) type [(0,1,0)]
```

on the high side.

Having produced an appropriate box, we iterate over its points, which means that we collect data from

```
(9,10,12)
(9,10,13)
```

on the low side and from

```
(8,10,12)
(8,10,13)
```

on the high side. But this is exactly what we have in equation (54), where $\bar{k}$ implies the summation over k and $k+1$, i.e., in this case, over $k = 12$ and $k = 13$.

The summation occurs upon multiplication of each term taken from *a_fineFlux* by *scale*.

In our case, looking again at equation (54), *scale* should be $\frac{1}{6\Delta X}$, where 1/3 comes from the time step and 1/2 comes from the division of *a_scale* by *m_nRefine*. This implies that we should set *a_scale* to be $\pm\frac{1}{3\Delta X}$.

23 ⟨Body of incrementFine 18⟩ +≡

```
for (regIt.begin( ); regIt.ok( ); ++regIt) {
  const IntVect &coarseIndex = regIt( );
  Box box(coarseIndex, coarseIndex);
  box.shiftHalf(regcomp, −1);
  IntVect refineVect(D_DECL(m_nRefine, m_nRefine, m_nRefine));
  box.refine(refineVect);
  if (a_sd ≡ Side::Lo) box.growLo(a_dir, −(m_nRefine − 1));
  else box.growHi(a_dir, −(m_nRefine − 1));
  BoxIterator fluxIt(box);
  for (fluxIt.begin( ); fluxIt.ok( ); ++fluxIt) {
    int src = a_srcInterval.begin( );
    int dest = a_dstInterval.begin( );
    for ( ; src ≤ a_srcInterval.end( ); ++src, ++dest)
      reg(coarseIndex, dest) += scale ∗ a_fineFlux(fluxIt( ), src);
  }
}
```

¶ This closes the function. Before returning we *shiftHalf* the input field *a_fineFlux* back into place.

24 ⟨Body of incrementFine 18⟩ +≡

```
a_fineFlux.shiftHalf(a_dir, −sign(a_sd));
```

5.3.1 The Saga of the Signs, Again

What exactly should we feed into *incrementFine*? Let us again inspect equations (43) and (44). These are the corrections to B_y and B_z that should be applied on the low side. The fine side contribution to the B_y correction on the low side of the B_x face is

$$\frac{\langle e_z(i,j+2,\bar{k})\rangle|_{T+\Delta T/6} + \langle e_z(i,j+2,\bar{k})\rangle|_{T+\Delta T/2} + \langle e_z(i,j+2,\bar{k})\rangle|_{T+5\Delta T/6}}{3}\frac{\Delta T}{\Delta X} \tag{55}$$

and the fine side contribution to the B_z correction on the low side of the B_x face is

$$-\frac{\langle e_y(i,\bar{j},k+2)\rangle|_{T+\Delta T/6} + \langle e_y(i,\bar{j},k+2)\rangle|_{T+\Delta T/2} + \langle e_y(i,\bar{j},k+2)\rangle|_{T+5\Delta T/6}}{3}\frac{\Delta T}{\Delta X} \tag{56}$$

Function *incrementFine* will automatically carry out the summations over $\bar{k}$ for (55) and $\bar{j}$ for (56) and the divisions by 2. But we have to take care of $\Delta T/(3\Delta X)$.

Inspection of the *incrementFine* code shows that there is no hocus pocus with signs here. The sign of *scale* and *a_scale* is not altered anywhere and the sum that goes into *reg*(*coarseIndex*, *dest*) is all addition.

Whereas the correction (55) is with the plus sign on the low side, it should be with the minus on the high side as we find in equation (34) and similarly for B_z on the high side, as we find in equation (35), where the correction should be entered with the plus. But let us recall that *refluxCurl* does take care of changing the correction sign depending on the side.

The actual correction that is implemented by *refluxCurl* is $-\mathit{local\_scale} * (\mathit{coar} + \mathit{fine})$, so the contribution of the fine register is $-\mathit{local\_scale} * \mathit{fine}$, which is going to be $-\mathit{a\_scale} * \mathit{fine}$ on the low side of the B_x face for the B_y correction, as shown in table 2, page 19. Let us make *a_scale* here ΔT.

To get the plus for the B_y correction on the low side of the B_x face we need to set *a_scale* on the call to *incrementFine* to $-1/(3\Delta X)$ on the low side and on the high side as well, because *refluxCurl* will take care of making the sign dependent on the side.

In other words, we feed the high side field to *incrementFine*, unlike was the case with *incrementCoarse*, where we fed the low side field. Why is this so? This is because both registers contribute their corrections in *refluxCurl* with the same sign, whereas the actual signs of the corrections contributed by coarse and fine registers in equations (43), (44), (34) and (35) were opposite.

6 Auxiliary Functions

Here we have three auxiliary functions, two of which we have encountered already, namely *index* and *getRegComp*, the third one, *setDefaultValues*, is used by the class constructors.

All three are short and do not require much explanation. *index* converts *dir* and *side* into a single number between 0 and 5, so that we can access the appropriate register slot. *getRegComp* finds, in a somewhat convoluted manner, the third dimension that is perpendicular to both *faceDir* and *edgeDir*.

It is here and *only* here that *SpaceDim* other than 3 will fail on the *assert* and abort the program. We should have something similar in the constructor.

26 ⟨Auxiliaries 26⟩ ≡
```
void LevelFluxRegisterEdge::setDefaultValues( )
{
  m_isDefined = false;
  m_nComp = −1;
  m_nRefine = −1;
}
int LevelFluxRegisterEdge::index(int dir, Side::LoHiSide side)
{
  return side ∗ SpaceDim + dir;
}
int LevelFluxRegisterEdge::getRegComp(const int &faceDir, const int &edgeDir)
{
  int regcomp;
  assert(SpaceDim ≡ 3);
  if (faceDir ≡ 0) {
    regcomp = 1;
    if (edgeDir ≡ 1) regcomp = 2;
  }
  else if (faceDir ≡ 1) {
    regcomp = 0;
    if (edgeDir ≡ 0) regcomp = 2;
  }
  else if (faceDir ≡ 2) {
    regcomp = 0;
    if (edgeDir ≡ 0) regcomp = 1;
  }
  return regcomp;
}
bool LevelFluxRegisterEdge::isDefined( ) const
{
  return m_isDefined;
}
void LevelFluxRegisterEdge::undefine( )
{
  m_isDefined = false;
}
```
This code is used in chunk 39.

7 Debugging Utilities

These five functions don't really do anything, since the calls to *dumpLDF*( ) are commented out and the function itself is not provided. They should be considered merely place holders.

```
27 ⟨Debugging Utilities 27⟩ ≡
  void LevelFluxRegisterEdge::dump( )
  {
    for (int idir = 0; idir < SpaceDim; idir++) {
      dumpLoCoar(idir);
      dumpLoFine(idir);
      dumpHiCoar(idir);
      dumpHiFine(idir);
    }
  }
  void LevelFluxRegisterEdge::dumpLoCoar(int a_idir)
  {
    const LevelData⟨FluxBox⟩ &loFabCoar = fabCoarse[index(a_idir, Side::Lo)];
    cout << "LoFabCoar::" << a_idir << endl;      /* dumpLDF(&loFabCoar); */
  }
  void LevelFluxRegisterEdge::dumpHiCoar(int a_idir)
  {
    const LevelData⟨FluxBox⟩ &hiFabCoar = fabCoarse[index(a_idir, Side::Hi)];
    cout << "HiFabCoar::" << a_idir << endl;      /* dumpLDF(&hiFabCoar); */
  }
  void LevelFluxRegisterEdge::dumpLoFine(int a_idir)
  {
    const LevelData⟨FluxBox⟩ &loFabFine = fabFine[index(a_idir, Side::Lo)];
    cout << "LoFabFine::" << a_idir << endl;      /* dumpLDF(&loFabFine); */
  }
  void LevelFluxRegisterEdge::dumpHiFine(int a_idir)
  {
    const LevelData⟨FluxBox⟩ &hiFabFine = fabFine[index(a_idir, Side::Hi)];
    cout << "HiFabFine::" << a_idir << endl;      /* dumpLDF(&hiFabFine); */
  }
```
This code is used in chunk 39.

8 The Class Constructor

The object construction for the class is deferred through various wrappers to function *define* that allocates space for and structures various protected variables, which we have already encountered, namely

fabCoarse the coarse register

fabFine the fine register

refluxLocations a list of cells for which we need to reflux, used by *refluxCurl*

coarToCoarMap a map that converts coarse level grid locations to the coarsened fine level grid locations

crseCopiers predefined copiers to be use when copying data between the fine register, *fabFine*, and a temporary register, *fineRegLocal*, used by *refluxCurl*

m_nComp internal copy of the number of components

m_nRefine internal copy of the refinement ratio

m_domainCoarse problem domain of the coarse level.

Apart from defining the structures, function *define* also fills *refluxLocations* and *coarToCoarMap*.

8.1 Define

define is a pretty horrible function that is some 275 lines long and convoluted. But it's easier to understand when it's split into chunks.

It's basic structure is a loop over face directions and sides. Within this loop there are two sub-loops: the first one iterates over the fine register boxes, which are related to, but not the same as the fine level boxes, and the second one iterates over the coarse grid boxes. The loop over the fine register boxes splits into three conditions depending on where the boxes reside with respect to the computational domain. The loop over the coarse grid boxes is the one that puts data into *refluxLocations* and *coarToCoarMap*. It also splits into two conditions depending on where the coarse grid boxes reside.

> A great deal of the function's complexity derives from its handling of periodic boundaries. It would be good to write a simpler and *cleaner* function that does not handle periodic boundaries and then perhaps wrap another class around it that does.

This version of *define* takes the following arguments.

const DisjointBoxLayout &*a_dbl* A disjoint box layout of the fine grid.

const DisjointBoxLayout &*a_dblCoarse* A disjoint box layout of the coarse grid.

const ProblemDomain &*a_dProblem* A fine grid version of the problem domain.

int *a_nRefine* Refinement ratio.

int *a_nComp* Number of components per cell side to curl-reflux—this is usually one, not three, e.g., in case of a $\boldsymbol{B}$ field, B_x, B_y and B_z all live on different sides.

The body of the function begins with various trivial operations. First we set *m_isDefined* to *true*, even though this is a little too early. We really should wait with it until the end. Then we check that the refinement ratio and the number of components are sound, and make sure that the problem domain is not empty.

a_nComp and *a_nRefine* are copied onto the protected class variables *m_nComp* and *m_nRefine*. The problem domain is coarsened and we make sure that its periodicity is the same as that of the coarse level disjoint box layout.

Then we size the array of copiers, but don't do anything to them yet, define the shift iterator, which will be needed to handle periodic boundary conditions, if such are present, and extract a box from the coarsened problem domain. We will test boxes of the fine and coarse level against this box to check if any stick out. This box is further conditioned depending on whether the problem domain is periodic. If it is, then the box is shortened in the direction of periodicity by one layer of cells. Any fine or coarse grid level box that sticks out of the domain box in a periodic direction, abuts the periodic boundary and so has to be treated specially.

Having dealt with these preliminaries we proceed to iterate over face directions and sides, which is deferred to the next module, ⟨ Iterate over Directions and Sides 30 ⟩.

29 ⟨ define 29 ⟩ ≡

```
void LevelFluxRegisterEdge::define(const DisjointBoxLayout &a_dbl, const DisjointBoxLayout
        &a_dblCoarse, const ProblemDomain &a_dProblem, int a_nRefine, int a_nComp)
{
  m_isDefined = true;
  assert(a_nRefine > 0);
  assert(a_nComp > 0);
  assert(¬a_dProblem.isEmpty());
  m_nComp = a_nComp;
  m_nRefine = a_nRefine;
  m_domainCoarse = coarsen(a_dProblem, a_nRefine);
  assert(a_dblCoarse.checkPeriodic(m_domainCoarse));
  crseCopiers.resize(SpaceDim * 2);

  ShiftIterator shiftIt = m_domainCoarse.shiftIterator();
  Box periodicTestBox(m_domainCoarse.domainBox());
```

```
    if (m_domainCoarse.isPeriodic()) {
      for (int idir = 0; idir < SpaceDim; idir++) {
        if (m_domainCoarse.isPeriodic(idir)) periodicTestBox.grow(idir, −1);
      }
    }
    ⟨ Iterate over Directions and Sides 30 ⟩
  }
```
This code is cited in chunk 32.
This code is used in chunk 39.

¶ Within this loop we allocate space for the fine and coarse registers and structure them.

We begin from the fine register. First we create temporary box layouts, *tmp* and *fineBoxes*. These are created just for this space dimension and for this side and they will be destroyed when we end this particular iteration. We initiate *tmp* to be a coarsened image of the fine grid disjoint box layout. Then we perform a very peculiar operation on it.

The standard Chombo operations *adjCellHi* and *adjCellLo* are defined for a **Box**, a **DisjointBoxLayout** and for a **ProblemDomain**. When applied to a **Box** they return a new cell centered **Box** of a given length (here the length is 1) and on the low or high side of the argument **Box** in a given direction *idir*.

For example, if

```
IntVect index1(D_DECL(4,5,6));
IntVect index2(D_DECL(6,7,8));
Box box(index1,index2);
Box newbox = adjCellHi(box, 0, 1);
```

then the two boxes print as follows:

```
box:    Box (4,5,6) to (6,7,8) type [(0,0,0)]
newbox: Box (7,5,6) to (7,7,8) type [(0,0,0)]
```

The *newbox* here is a one-cell thick box that is drawn *to the side* of the original box.

When *adjCellHi* and *adjCellLo* are applied to a **DisjointBoxLayout** or to a **ProblemDomain**, they do it to *every* box of the layout or the domain. In effect we end up with a new, quite particular disjoint box layout that is going to stick out by one cell in the direction *idir* and on the side *side*( ), and that is going to be made just of the one-cell thick boundaries of all the boxes in the original layout. This is then written on *fineBoxes*, and this box layout is then used to define the fine register *fabFine* for this direction, for this side, and for the number of components passed in *a_nComp*. An example of such a *fineBoxes* layout is shown in Figure 3.

Next we define a structure that will hold information about cell faces that are right on the fine-coarse boundary, as opposed to cell faces that abut other cells of the fine grid. This structure is *coarseFineIVSVect* and it is a **Vector** of **Vector**s of **IntVectSet**s, where the first **Vector** has *SpaceDim* slots and the second vector's length is the number of boxes in the *fineBoxes* layout.

Now we are ready to begin iteration over the *fineBoxes*, and the details of what we do here are deferred to module ⟨ Iterate over the Fine Register Boxes 31 ⟩. Within this module we will collect the boundary points in *coarseFineIVSVect* by iterating over all boxes of the original coarsened fine box layout and subtracting them from *fineBoxes*. What is going to be left will be the *fineBoxes* cells that dangle just outside of the fine level box layout, i.e., the fine/coarse boundary—for this face direction and this side.

Having emerged from this loop we allocate space to *refluxLocations* and to *coarToCoarMap* for this face direction and for this side, and then plunge into the second loop, this one over the coarse grid boxes. The discussion of what goes on within it is deferred to module ⟨ Iterate over the Coarse Grid Boxes 35 ⟩.

30 ⟨ Iterate over Directions and Sides 30 ⟩ ≡
```
  SideIterator side;
  for (int idir = 0; idir < SpaceDim; ++idir) {
    for (side.begin(); side.ok(); ++side) {
      DisjointBoxLayout fineBoxes, tmp;
```

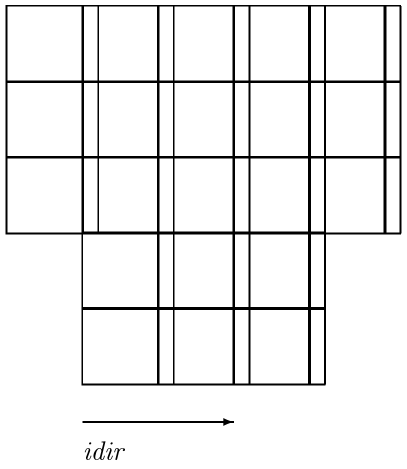

Figure 3: Example of the *fabFine* grid for $idir = 0$ and $side() = \mathbf{Side}::Hi$. The grid is made of the narrow boxes that appear to the right of the original square boxes of the fine level grid.

```
        coarsen(tmp, a_dbl, m_nRefine);
        if (side() ≡ Side::Lo) adjCellLo(fineBoxes, tmp, idir, 1);
        else adjCellHi(fineBoxes, tmp, idir, 1);
        fabFine[index(idir, side())].define(fineBoxes, a_nComp);
        Vector⟨Vector⟨IntVectSet⟩⟩ coarseFineIVSVect(SpaceDim);
        for (int faceDir = 0; faceDir < SpaceDim; faceDir++)
          coarseFineIVSVect[faceDir].resize(fineBoxes.size());
        ⟨Iterate over the Fine Register Boxes 31⟩
        LayoutData⟨Vector⟨Vector⟨IntVectSet⟩⟩⟩ &ivsetsVect = refluxLocations[index(idir, side())];
        ivsetsVect.define(a_dblCoarse);
        LayoutData⟨Vector⟨DataIndex⟩⟩ &mapsV = coarToCoarMap[index(idir, side())];
        mapsV.define(a_dblCoarse);
        ⟨Iterate over the Coarse Grid Boxes 35⟩
      }
    }
```

This code is cited in chunks 5, 29, and 35.

This code is used in chunk 29.

¶ Within this loop over all *fineBoxes* of the fine register, for this *idir* and *side*( ), we loop over all directions *perpendicular* to *idir*. We also number all the *fineBoxes* with *i* and use the box number to index **IntVectSet**s of *coarseFineIVSVect* as well.

For each *faceDir* (different from *idir*) we extract the **Box** from *m_domainCoarse* and we first check if the fine register Box pointed to by *it2*( ) is contained within the domain box. If it is we execute the ⟨Fine Register Box in the Coarsened Domain's Box 32⟩ module.

If this is not the case, we check if the coarse domain *contains* the fine register box *it2*( ). What is the difference between one and the other? At first glance, it looks like the same thing.

The difference is that if a domain is periodic in some direction, then it is *infinite* in this direction. Therefore any value of a cell index in this direction fits in the domain, and the cell is contained in the domain if the remaining indexes fit too. On the other hand, if we extract a **Box** from the domain, and the **Box** is *always* finite, and check if the **Box** contains the cell, then if the cell's coordinates protrude in some direction, the cell is not in the **Box**, and this is it. However, in this latter case the box may still abut the periodic boundary and this, of course, must be handled.

Anyhow, we find that this second clause is for situations when the domain is periodic and the fine register box, for this *idir* and this *side*( ), sticks out from the domain box in the periodic direction. This is handled by module ⟨ Fine Register Box in the Periodically Extended Domain 33 ⟩.

Finally, if the **Box** fits in neither of the two conditions, it means that it is really outside the computational domain, and this is handled by module ⟨ Fine Register Box Outside 34 ⟩.

31 ⟨ Iterate over the Fine Register Boxes 31 ⟩ ≡
```
LayoutIterator it = fineBoxes.layoutIterator( );
LayoutIterator it2 = fineBoxes.layoutIterator( );
int i = 0;
for (it2.begin( ); it2.ok( ); ++i, ++it2) {
  for (int faceDir = 0; faceDir < SpaceDim; faceDir++) {
    if (faceDir ≠ idir) {
      IntVectSet &ivs = coarseFineIVSVect[faceDir][i];
      const Box &compDomBox = m_domainCoarse.domainBox( );
      if (compDomBox.contains(fineBoxes.get(it2( )))) {
        ⟨ Fine Register Box in the Coarsened Domain's Box 32 ⟩
      }
      else if (m_domainCoarse.contains(fineBoxes.get(it2( )))) {
        ⟨ Fine Register Box in the Periodically Extended Domain 33 ⟩
      }
      else {
        ⟨ Fine Register Box Outside 34 ⟩
      }
    }
  }
}
```
This code is cited in chunks 30, 35, and 36.
This code is used in chunk 30.

¶ Inside this ⟨ Fine Register Box in the Coarsened Domain's Box 32 ⟩ module we get the fine register box pointed to by *it2*( ), attach surrounding nodes in the *faceDir* direction, and this will make it node-centered in this direction too, so we half-shift it in the *faceDir* direction by one half to make it again cell-centered, and pack all its points into *ivs*, which is a C++ alias for *coarseFineIVSVect*[*faceDir*][*i*], where *i* is the number of the **Box** in the fine register's box layout.

If we now subtract all the boxes of the original fine level box layout (the square ones in Figure 3) from all the boxes or the fine register box layout (the narrow vertical ones in Figure 3), what will be left in *ivs* will be only the protruding one-cell thick boxes on the right of the whole layout, and this is how we identify the boundary *for this direction and for this side.*

This is indeed what we do here.

We enter another loop over *it*( ). For each box of the fine level box layout pointed to by *it*( ) we coarsen it, then we attach surrounding nodes in the *faceDir* direction, half-shift by 1 in the *faceDir* direction to make it again cell centered and subtract all its points from *ivs*.

If the domain is periodic and the coarsened fine level box abuts the domain boundary—and we test for it by checking if it fits in the *periodicTestBox* we have constructed in module ⟨ define 29 ⟩—we go through a procedure that we have discussed already in our weave of the **PiecewiseLinearFillPatchFace** class: we wrap the box around the periodic boundary, for every direction in which periodicity occurs, subtract those points from *ivs*, then shift the box back in place.

32 ⟨ Fine Register Box in the Coarsened Domain's Box 32 ⟩ ≡
```
Box fineFaceBox = fineBoxes.get(it2( ));
fineFaceBox.surroundingNodes(faceDir);
fineFaceBox.shiftHalf(faceDir, 1);
ivs.define(fineFaceBox);
for (it.begin( ); it.ok( ); ++it) {
    Box coarsenedFineBox = coarsen(a_dbl[it()], a_nRefine);
    coarsenedFineBox.surroundingNodes(faceDir);
    coarsenedFineBox.shiftHalf(faceDir, 1);
    ivs −= coarsenedFineBox;
    if (m_domainCoarse.isPeriodic() ∧ ¬periodicTestBox.contains(coarsenedFineBox) ∧
          ¬periodicTestBox.contains(fineBoxes.get(it2()))) {
      IntVect shiftMult(m_domainCoarse.domainBox().size());
      for (shiftIt.begin(); shiftIt.ok(); ++shiftIt) {
        IntVect shiftVect = shiftIt() ∗ shiftMult;
        coarsenedFineBox.shift(shiftVect);
        ivs −= coarsenedFineBox;
        coarsenedFineBox.shift(−shiftVect);
      }
    }
  }
```
This code is cited in chunks 31, 32, 33, and 36.

This code is used in chunk 31.

¶ Here we have the situation when the fine register box pointed to by *it2*( ) sticks out of the problem domain box, but in a direction in which the domain is periodic. Such a box is still in the domain. So what we do here is as follows. First we make a copy of it on *regBox*. Then we shift the *regBox* in the periodic direction so as to put it into the box of the domain. Once it is there we just repeat all the operations carried out in module ⟨ Fine Register Box in the Coarsened Domain's Box 32 ⟩ on it, i.e., we put all its points in *ivs* and then subtract all boxes of the fine level box layout from *ivs*.

33 ⟨ Fine Register Box in the Periodically Extended Domain 33 ⟩ ≡
```
  IntVect shiftMult(compDomBox.size());
  Box regBox = fineBoxes.get(it2());
  for (shiftIt.begin(); shiftIt.ok(); ++shiftIt) {
    IntVect shiftVect(shiftIt() ∗ shiftMult);
    regBox.shift(shiftVect);
    if (compDomBox.contains(regBox)) {
      assert(ivs.isEmpty());
      Box fineFaceBox = regBox;
      fineFaceBox.surroundingNodes(faceDir);
      fineFaceBox.shiftHalf(faceDir, 1);
      ivs.define(fineFaceBox);
      ShiftIterator shiftIt2 = m_domainCoarse.shiftIterator();
      for (it.begin(); it.ok(); ++it) {
        Box coarsenedFineBox = coarsen(a_dbl[it()], a_nRefine);
        coarsenedFineBox.surroundingNodes(faceDir);
        coarsenedFineBox.shiftHalf(faceDir, 1);
        ivs −= coarsenedFineBox;
        if (¬m_domainCoarse.isPeriodic() ∧ ¬periodicTestBox.contains(coarsenedFineBox) ∧
              ¬periodicTestBox.contains(fineBoxes.get(it2()))) {
          IntVect shiftMult(m_domainCoarse.domainBox().size());
          for (shiftIt2.begin(); shiftIt2.ok(); ++shiftIt2) {
            IntVect shiftVect = shiftMult ∗ shiftIt2();
            coarsenedFineBox.shift(shiftVect);
            ivs −= coarsenedFineBox;
            coarsenedFineBox.shift(−shiftVect);
          }
```

```
            }
        }
    }
}
```

This code is cited in chunks 31 and 36.

This code is used in chunk 31.

¶ This last case corresponds to the situation when a fine register box sticks out of the computational domain, but is not in the domain's periodic direction either. Such a box is obviously a fine/coarse boundary box and so, all its points go into *ivs* and nothing gets subtracted.

34 ⟨ Fine Register Box Outside 34 ⟩ ≡

```
Box fineFaceBox = fineBoxes.get(it2());
fineFaceBox.surroundingNodes(faceDir);
fineFaceBox.shiftHalf(faceDir, 1);
ivs.define(fineFaceBox);
```

This code is cited in chunks 31 and 36.

This code is used in chunk 31.

¶ Now, let us go back to module ⟨ Iterate over Directions and Sides 30 ⟩.

After we had finished with ⟨ Iterate over the Fine Register Boxes 31 ⟩, we picked up the slice of *refluxLocations* for this face direction and for this side, this being a **LayoutData** of **Vector**⟨**Vector**⟨**IntVectSet**⟩⟩, and defined it over the coarse level box layout, *a_dblCoarse*. Having done so we named it locally *ivsetsVect*.

We had done similarly with the corresponding slice of the conversion map *coarToCoarMap*, this being a **LayoutData** of **Vector**⟨**DataIndex**⟩, and defined it over the coarse level box layout, as well. Having done so we named it locally *mapsV*.

In summary we'll see the following two references in the code below:

ivsetsVect, which is a local reference to *refluxLocations*, and

mapsV, which is a local reference to *coarToCoarMap*.

The **for** loop that unfolds now is over all boxes of the coarse level box layout. Within this loop we will locate coarse grid boxes for which we will need to do the refluxing, i.e., the fine/coarse border boxes on the coarse grid side. The border boxes will be collected on the *coarseBoxes* layout, which will then be used to define the coarse register and the **Copier**s.

Since we are at it, a word of explanation about what a **Copier** does and how it is defined. a **Copier** predefines data movement patterns between two box layouts, in this case, the fine register box layout, *fineBoxes*, and the coarse register box layout, *coarseBoxes*. On this occasion additional specifications pertaining to the intended copy operation may be provided, for example, whether *exchange* should be invoked after the copy operation, or whether ghost cells of the destination box layout should be filled, or whether the copy operation should be restricted to the valid **ProblemDomain** only, or whether the data should be taken from the source level ghost cells.

Once defined, a **Copier** is passed to the **LevelData**::*copyTo* method as an argument to speed up the operation. Otherwise *copyTo* would have to redo, what the **Copier** does, every time it is invoked.

Here, when defining the array of *crseCopiers*, which is indexed by this face direction and this side, we provide source (*fineBoxes*) and target (*coarseBoxes*) box layouts, and tell the **Copier** to ignore ghost data by making the third argument **IntVect**::*Zero*.

But let us get back to the loop.

Within the **for** loop that runs over all boxes of the coarse level box layout we pick up the coarse box and then loop over all boxes of the fine register, which is what *it*( ) points to, as has been defined above in module ⟨ Iterate over the Fine Register Boxes 31 ⟩.

Now, if the fine register box pointed to by *it*( ) and the coarse level box pointed to by *dit*( ) intersect, we execute ⟨ The Coarse Box Intersects with the Fine Box 36 ⟩ module, otherwise, if the domain is periodic, but the *periodicTestBox* contains neither the fine register box, nor the coarse level box, we execute the ⟨ Both Boxes Dangle Beyong the Edge 37 ⟩ module, since this situation requires some box shifting and taking periodic boundaries into account.

Before we get to the loop over the fine register boxes though we check if the coarse register box we point to currently lives on the same *processor* on which we do these computations. In a parallel setting boxes of a disjoint box layout live on different processors and each processor looks after its own. So, only if this is *our* box, we size the **Vector** of *refluxLocations* (here called *ivsetsVect*) to be *SpaceDim* long.

35 ⟨ Iterate over the Coarse Grid Boxes 35 ⟩ ≡

```
DisjointBoxLayout coarseBoxes;
LayoutIterator dit = a_dblCoarse.layoutIterator();
for (dit.begin(); dit.ok(); ++dit) {
  unsigned int thisproc = a_dblCoarse.procID(dit());
  if (thisproc ≡ procID()) {
    ivsetsVect[DataIndex(dit())].resize(SpaceDim);
  }
  const Box &coarseBox = a_dblCoarse[dit()];
  i = 0;
  int count = 0;
  for (it.begin(); it.ok(); ++it, ++i) {
    Box regBox = fineBoxes.get(it());
    if (regBox.intersectsNotEmpty(coarseBox)) {
      ⟨ The Coarse Box Intersects with the Fine Box 36 ⟩
    }
    else if (m_domainCoarse.isPeriodic() ∧ ¬periodicTestBox.contains(regBox) ∧
          ¬periodicTestBox.contains(coarseBox)) {
      ⟨ Both Boxes Dangle Beyong the Edge 37 ⟩
    }
  }
}
coarseBoxes.close();
fabCoarse[index(idir, side())].define(coarseBoxes, a_nComp);
crseCopiers[index(idir, side())].define(fineBoxes, coarseBoxes, IntVect::Zero);
```

This code is cited in chunks 30 and 36.

This code is used in chunk 30.

¶ Here we have a straightforward situation: the fine register box and the coarse grid box intersect.

The fine register box is represented here by *regBox*, as has been defined in module ⟨ Iterate over the Coarse Grid Boxes 35 ⟩. The operation &= modifies *regBox* by intersection with *coarseBox*, so, from now on *regBox* contains cells that are common to the original fine register box and to the coarse level box.

We find the processor that this coarse level box lives on and add the modified *regBox* to the *coarseBoxes* layout, originally empty when created, assigning it to the same processor.

If the coarse level box processor happens to be the one we are currently on, we have to do more work.

The box layout operation *addBox* adds a box to the layout as a side effect. Its return is a **DataIndex**, as would have been returned by the corresponding **DataIterator** for this box layout. And this is rather convenient, because we can now push this **DataIndex** on *mapsV*, which, as we have pointed out in ⟨ Iterate over the Coarse Grid Boxes 35 ⟩, is our local name for the slot of *coarToCoarMap* that corresponds to this face direction and this side.

The following **for** loop, over the face directions that are different from *idir* trims the content of *refluxLocations* for this direction and side by intersecting it with *regBox* that has been stretched in the *faceDir* direction to add the *surroundingNodes*. The manipulations that *regBox* is subjected to here are easy to understand, but we need to explain the two operations on the *refluxLocations* in more detail.

The first thing we do is to push the points that we have collected on this slot of *coarseFineIVSVect* (numbered by *faceDir* and *i*, where *i* is the number of the fine register box) in module ⟨ Iterate over the Fine Register Boxes 31 ⟩. There, to remind the reader, we have collected boundary cells—for this face direction and this side—on the *coarseFineIVSVect*'s local alias *ivs*, see modules ⟨ Fine Register Box in the Coarsened Domain's Box 32 ⟩, ⟨ Fine Register Box in the Periodically Extended Domain 33 ⟩, and ⟨ Fine Register Box Outside 34 ⟩. So, this is how the *refluxLocations* gets filled with cells. But we still need to trim them by intersecting with appropriately stretched *regBox*.

Let us observe that whereas *i* numbers the fine register boxes, *count* counts the number of intersections with the fine register boxes for the coarse level box pointed to by *dit*( )—beginning with 1. Everything that has been accumulated in *coarseFineIVSVect* from a given fine register box *i* goes into *ivsetsVect*[**DataIndex**(*dit*( ))][*faceDir*] including cells that *do not overlap* with the coarse level box *dit*( ). For the *count* intersection though, we want only those cells that belong both to the coarse level box *dit*( ) and to the fine register box *it*( ). The corresponding slot in *ivsetsVect* is [*faceDir*][*count* − 1], and so we take the content of this slot, call it *reducedIVS*, and intersect with appropriately stretched *regBox* to eliminate the unwanted cells.

36 ⟨ The Coarse Box Intersects with the Fine Box 36 ⟩ ≡
```
regBox &= coarseBox;
++count;
unsigned int proc = a_dblCoarse.procID(dit());
DataIndex index = coarseBoxes.addBox(regBox, proc);
if (proc ≡ procID()) {
  mapsV[DataIndex(dit())].push_back(index);
  for (int faceDir = 0; faceDir < SpaceDim; faceDir++) {
    if (faceDir ≠ idir) {
      ivsetsVect[DataIndex(dit())][faceDir].push_back(coarseFineIVSVect[faceDir][i]);
      IntVectSet *reducedIVS = &(ivsetsVect[DataIndex(dit())][faceDir][count − 1]);
      Box faceRegBox = regBox;
      faceRegBox.surroundingNodes(faceDir);
      faceRegBox.shiftHalf(faceDir, 1);
      (*reducedIVS) &= faceRegBox;
    }
  }
}
```
This code is cited in chunks 35 and 37.
This code is used in chunk 35.

¶ Here we have the situation when both the fine register box and the coarse grid box dangle beyond the edge of the computational domain's box and the domain is periodic, which means that the boxes in question may be within the domain after all—to remind the reader, a periodic domain is infinite in the direction of its periodicity.

If this is the case, we shift the *regBox*, i.e., the fine register box pointed to by *it*( ), into the domain's box and then proceed *exactly* as we have in ⟨ The Coarse Box Intersects with the Fine Box 36 ⟩ module.

Having done so we shift the *regBox* back to where it was and then shift it in another periodic direction, and so on, until we have run through all periodic directions of the domain.

37 ⟨ Both Boxes Dangle Beyong the Edge 37 ⟩ ≡
```
IntVect shiftMult(m_domainCoarse.domainBox().size());
for (shiftIt.begin(); shiftIt.ok(); ++shiftIt) {
  IntVect shiftVect = shiftMult * shiftIt();
  regBox.shift(shiftVect);
  if (regBox.intersectsNotEmpty(coarseBox)) {
    regBox &= coarseBox;
    ++count;
    unsigned int proc = a_dblCoarse.procID(dit());
    DataIndex index = coarseBoxes.addBox(regBox, proc);
```

```
      if (proc ≡ procID()) {
        mapsV[DataIndex(dit())].push_back(index);
        for (int faceDir = 0; faceDir < SpaceDim; faceDir++) {
          if (faceDir ≠ idir) {
            ivsetsVect[DataIndex(dit())][faceDir].push_back(coarseFineIVSVect[faceDir][i]);
            IntVectSet *reducedIVS = &(ivsetsVect[DataIndex(dit())][faceDir][count − 1]);
            Box faceRegBox = regBox;
            faceRegBox.surroundingNodes(faceDir);
            faceRegBox.shiftHalf(faceDir, 1);
            (*reducedIVS) &= faceRegBox;
          }
        }
      }
    }
    regBox.shift(−shiftVect);
  }
```
This code is cited in chunk 35.

This code is used in chunk 35.

8.2 Wrappers

And this is all there is to *define*, apart from wrappers, i.e., explicit class object construction calls that, in turn, call *define* to do the job. These are simple enough, so that we don't have to comment on them. *define* itself may be called with a **Box** instead of **ProblemDomain**, in which case, the **Box** is converted to a **ProblemDomain** and *define* called with its primary interface.

38 ⟨Wrappers 38⟩ ≡

```
LevelFluxRegisterEdge::LevelFluxRegisterEdge(const DisjointBoxLayout &a_dblFine, const
        DisjointBoxLayout &a_dblCoar, const Box &a_dProblem, int a_nRefine, int a_nComp)
{
  setDefaultValues();
  ProblemDomain physdomain(a_dProblem);
  define(a_dblFine, a_dblCoar, physdomain, a_nRefine, a_nComp);
}
LevelFluxRegisterEdge::LevelFluxRegisterEdge(const DisjointBoxLayout &a_dblFine, const
        DisjointBoxLayout &a_dblCoar, const ProblemDomain &a_dProblem, int a_nRefine, int
        a_nComp)
{
  setDefaultValues();
  define(a_dblFine, a_dblCoar, a_dProblem, a_nRefine, a_nComp);
}
void LevelFluxRegisterEdge::define(const DisjointBoxLayout &a_dbl, const DisjointBoxLayout
        &a_dblCoarse, const Box &a_dProblem, int a_nRefine, int a_nComp)
{
  ProblemDomain physdomain(a_dProblem);
  define(a_dbl, a_dblCoarse, physdomain, a_nRefine, a_nComp);
}
LevelFluxRegisterEdge::LevelFluxRegisterEdge()
{
  setDefaultValues();
}
LevelFluxRegisterEdge::~LevelFluxRegisterEdge()
{ }
```

This code is used in chunk 39.

9 Putting it all together

Here we put it all together to provide instructions to *tangle* about how to generate the code.

39
```
#include "LevelFluxRegisterEdge.H"
#include "LayoutIterator.H"
#include "DebugOut.H"
#include "Copier.H"
#include "parstream.H"
  ⟨ Auxiliaries 26 ⟩
  ⟨ Wrappers 38 ⟩
  ⟨ define 29 ⟩
  ⟨ setToZero 11 ⟩
  ⟨ incrementCoarse 12 ⟩
  ⟨ incrementFine 17 ⟩
  ⟨ refluxCurl 4 ⟩
  ⟨ Debugging Utilities 27 ⟩
```

10 The Header File

For completeness we add the class header file, so that it can be regenerated by tangle together with the cpp file.

```
40 ⟨Reflux.H 40⟩ ≡
#ifndef _LEVELFLUXREGISTEREDGE_H_
# define _LEVELFLUXREGISTEREDGE_H_
#include "REAL.H"
#include "Vector.H"
#include "FluxBox.H"
#include "IntVectSet.H"
#include "CFStencil.H"
#include "LoHiSide.H"
#include "LevelData.H"
#include "LayoutData.H"
  class LevelFluxRegisterEdge {
  public:
    LevelFluxRegisterEdge( );
    LevelFluxRegisterEdge(const DisjointBoxLayout &a_dbl, const DisjointBoxLayout
        &a_dblCoarse, const Box &a_dProblem, int a_nRefine, int a_nComp);
    LevelFluxRegisterEdge(const DisjointBoxLayout &a_dbl, const DisjointBoxLayout
        &a_dblCoarse, const ProblemDomain &a_dProblem, int a_nRefine, int a_nComp);
    ~LevelFluxRegisterEdge( );
    void define(const DisjointBoxLayout &a_dbl, const DisjointBoxLayout &a_dblCoarse, const Box
        &a_dProblem, int a_nRefine, int a_nComp);
    void define(const DisjointBoxLayout &a_dbl, const DisjointBoxLayout &a_dblCoarse, const
        ProblemDomain &a_dProblem, int a_nRefine, int a_nComp);
    void undefine( );
    void setToZero( );
    void incrementCoarse(FArrayBox &a_coarseFlux, Real a_scale, const DataIndex
        &a_coarseDataIndex, const Interval &a_srcInterval, const Interval &a_dstInterval, int a_dir);
    void incrementFine(FArrayBox &a_fineFlux, Real a_scale, const DataIndex &a_fineDataIndex, const
        Interval &a_srcInterval, const Interval &a_dstInterval, int a_dir, Side::LoHiSide a_sd);
    void refluxCurl(LevelData⟨FluxBox⟩ &a_uCoarse, Real a_scale);
    bool isDefined( ) const;
    void dump( );
    void dumpLoCoar(int idir);
    void dumpHiCoar(int idir);
    void dumpLoFine(int idir);
    void dumpHiFine(int idir);
  protected:
    static int index(int dir, Side::LoHiSide side);
    static int getRegComp(const int &faceDir, const int &edgeDir);
    void setDefaultValues( );
    LevelData⟨FluxBox⟩ fabCoarse[SpaceDim * 2];
    LevelData⟨FluxBox⟩ fabFine[SpaceDim * 2];
    Vector⟨Copier⟩ crseCopiers;
    LayoutData⟨Vector⟨Vector⟨IntVectSet⟩⟩⟩ refluxLocations[SpaceDim * 2];
    LayoutData⟨Vector⟨DataIndex⟩⟩ coarToCoarMap[SpaceDim * 2];
    bool m_isDefined;
    int m_nComp;
    int m_nRefine;
    ProblemDomain m_domainCoarse;
  private:
    void operator=(const LevelFluxRegisterEdge &);
    LevelFluxRegisterEdge(const LevelFluxRegisterEdge &);
  };
#endif
```

11 Closing Comments

The class **LevelFluxRegisterEdge** was designed some 5 or 6 years ago, and hasn't been updated since, whereas its simpler sibling, **LevelFluxRegister**, has been revised and optimised in many ways, this being a central class of the AMR method when applied to Computational Fluid Dynamics. This accounts for various minor glitches, dutifully flagged with the "dangerous bend" signs on the margins of this document, as well as possible inefficiencies.

The location and initialization of fine/coarse boundary structures, as discussed in section 8, is a costly endeavour and so it should not be repeated while the grids don't change. In case a program does not evolve the levels at all, it should be done right at the beginning of the execution and then only the *refluxCurl* invoked.

There are two factors here that make the code hard to read.

The first one, as we have remarked above, is its coverage of periodic boundaries. The second one is the need to account for face mounted nodes, hence various "hacks" such as stretching the boxes and half-shifts.

This second problem suggests that Chombo itself should provide more and better facilities to support data mounts other than at cell centers, so that the "hacks" deployed in this and in the "Prolongate" codes would not be needed. The first problem could be addressed by simply chopping out all parts of the code that address periodic boundaries. These changes would make the code much simpler and thus easier to manage, debug and optimize.

Does it all really work without a glitch? The code has been thoroughly tested and then used in MHD simulations, so, yes, it does work.

But if additional tests are needed, the debugging utilities provided as place-holders currently, should be enhanced and made to print register contents—both cell locations and data accumulated within the cells. This should be done both for direct printing and for HDF5 output.

References

[1] Dinshaw S. Balsara, “Divergence-Free Adaptive Mesh Refinement for Magnetohydrodynamics”, Journal of Computational Physics, vol. 174, pp. 614–648, 2001.

[2] Zdzisław Meglicki, Stephen K. Gray and Boyana Norris, “Multigrid FDTD with Chombo”, Computer Physics Communications, Available online 13 October 2006, doi:10.1016/j.cpc.2006.08.008

Index

Here is a list of the identifiers used, and the chunks where they appear. Underlined entries indicate the place of definition.

List of Refinements